Radio Shack
TRS-80
COMPUTER PRODUCTS
Model 4

TRS-80® PASCAL

Cat. No.
26-2212

Radio Shack®
TRS-80
COMPUTER PRODUCTS

SOFTWARE LIBRARY

CUSTOM MANUFACTURED IN USA BY RADIO SHACK, A DIVISION OF TANDY CORPORATION

TRS-80® Pascal

Radio Shack®
A DIVISION OF TANDY CORPORATION
FORT WORTH, TEXAS 76102

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.

TRS-80® Pascal

COPYRIGHT NOTICES

Model 4 TRS-80 Pascal Manual
Copyright 1983 by Alcor Systems
Licensed to Tandy Corporation
All rights reserved

Reproduction or use, without express written permission from Tandy Corporation and Alcor Systems of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure accuracy, Tandy Corporation and Alcor Systems assume no liability resulting from any errors or omissions in this manual or from the use of the information obtained herein.

Model 4 TRS-80 Pascal Software
Copyright 1983 by Alcor Systems
Licensed to Tandy Corporation
All rights reserved

TRSDOS Version 6 Operating System
Copyright 1983 by Logical Systems
Licensed to Tandy Corporation
All rights reserved

INTRODUCTION

Congratulations on the purchase of your model 4 TRS-80 Pascal programming system. TRS-80 Pascal is a powerful language system that will increase your productivity as a programmer.

The TRS-80 Pascal system is excellent for educational instruction because it is a complete implementation of the language as defined by its creator, Niklaus Wirth. In addition, some very important language extensions have been included to make it ideal as a professional program development system.

Included with the TRS-80 Pascal System is a very powerful, programmable full screen text editor. The editor characteristics may be easily changed to suit your personal preferences. You can map editor commands to the model 4 keyboard as desired and you can define your own editor commands.

The TRS-80 Pascal compiler generates a very efficient and compact object code. Programs developed with TRS-80 Pascal will typically execute between 10 and 50 times faster than equivalent programs developed with interpreted BASIC. The compact size of the object code allows you to develop very large programs without the need to resort to overlays or chaining.

IMPORTANT NOTE FOR MODEL 4 TRS-80 PASCAL USERS (Catalog Number 26-2212)

It is important to note that when using TRS-80 Pascal with the Model 4, the minimum system requirements are as follows :

64K Random Access Memory
Two Disk Drives
TRSDOS Version 6 Operating System

A printer capable of printing at least 80 columns per line and connecting cable are also recommended.

When using TRS-80 Pascal, a disk containing the TRSDOS Version 6 operating system must be in drive 0. None of the Model 4 TRS-80 Pascal disks contain an operating system. The TRSDOS Version 6 operating system must be supplied by the user.

INFORMATION FOR FIRST TIME USERS

If you are using this manual for the first time and would like to try writing a program in Pascal right away, then we suggest that after reading the Beginners Guide you go right to the Tutorial (section 4).

The Beginners Guide (pages 6 and 7) gives instructions for using the SAMPLE/EDT file on your disk. The SAMPLE/EDT file contains simple edit/key functions. When this file is renamed to SETUP/EDT, the editor will recognize the Model 4 arrow keys as commands that move the cursor around on the screen. Some of the other keys used to execute commands are also different from those described in the Editor Manual. Before renaming this file, be sure to make a backup of the disk labeled SYSTEM1 which is described on page 4 of the Beginners Guide. Label the backup so that you can distinguish it from the SYSTEM1 disk.

The editor is used for creating your Pascal programs. A few necessary edit commands will aid you in creating your first program.

1. From TRSDOS Ready, type edit and press <ENTER>.

2. While holding down the CLEAR key, press the N key repeatedly to insert blank lines into the editor's text buffer.

3. You save your program by pressing the F1 key and then at the <> prompt, type EXIT. The response to the prompt <EXIT>FILE: is the name you choose plus the extension /PCL and the drive number in the form :d.

   Example: DATABASE/PCL:1

   The response to the prompt <EXIT>BACKUP? is <ENTER>.

The editor has many more commands in addition to the ones described in the Beginners Guide. The Editor Manual describes the complete set of editor commands. The editor also has a command called HELP. This command is executed by pressing the F1 key and at the <> prompt typing HELP. At the <HELP>SUBJECT prompt, type one of the following:

HELP KEY CMD

The HELP subject displays information about the other two subjects, KEY and CMD. The KEY subject displays the keyboard layout of editor commands. The CMD subject displays the complete list of editor commands in alphabetic order. Each command has a two character mnemonic that may be used to execute the command. Simply press the F1 key and at the <> prompt, type the two character mnemonic.

NOTICE TO PROGRAMMERS

By purchase of the software product described in this manual, you have obtained a license to duplicate the supplied disk files only as necessary for personal use on your Model 4 computer. None of the supplied files may be reproduced for resale.

If you intend to sell application programs developed using TRS-80 Pascal, you must follow the procedure below to avoid violation of this license and of copyright laws.

1. Use the PASCAL compiler to translate the the application program to object code.

2. Use the LINKLOAD utility to link the object code with the TRS-80 Pascal runtime support and build a stand alone, executable command file (/CMD extension).

3. The executable command file may be copied and sold with no royalty payments required. However, all programs sold must document the fact that they contain TRS-80 PASCAL RUNTIME SUPPORT.

HOW TRS-80 PASCAL WORKS

TRS-80 Pascal is a compiled language. This means that programs must first be translated to object format before they may be executed.

The first step in developing a program is to enter the program into the computer and save it to a disk file. A full screen text editor (EDIT/CMD) is supplied to allow you to create your programs.

The second step is to compile the program. There are two versions of the compiler which are supplied for translating your programs to object format. One version is non-overlayed (PASCAL/CMD) while the other is overlayed (PASCALB/CMD). You should use the non-overlayed compiler most of the time because it is faster. However, when compiling very large programs, it may be necessary to use the overlayed compiler.

The third step is to execute the program. There is a run utility (RUNP/CMD) supplied which will execute your compiled programs. The run utility loads and executes object format files created by the compiler.

The linking loader utility (LINKLOAD/CMD) must be used to execute programs which have been split into separately compiled segments. The linking loader loads one or more object format files and links them into a single executable program. It has the ability to execute the program directly or to build an executable command file.

Optional optimizations may be performed to decrease the size of a compiled program or to increase its execution speed. The optimize utility (OPTIMIZE/CMD) reduces the the size of an object format file by 10 to 30 percent. The codegen utility (CODEGEN/CMD) translates an object format file into machine instructions which increases execution speed 3 to 5 times.

AN OVERVIEW OF THE TRS-80 PASCAL MANUAL

There are seven sections to this manual. We suggest that you read through the Beginners Guide carefully. The seven sections are:

1/ BEGINNERS GUIDE
(1) Takes you through the steps of backing up the system.
(2) Leads you through the steps of entering and executing a a simple Pascal program.
(3) Introductory trouble shooting guide.

2/ EDITOR MANUAL
Shows how to use the Blaise II text editor in detail.

3/ SYSTEM IMPLEMENTATION MANUAL
Gives specific information on the TRS-80 Model 4 implementation of Pascal.

Included is more detailed information on:

(1) Compiling and executing programs.
(2) TRS-80 Pascal memory usage.
(3) Using the external library of procedures and functions. (Graphics, keyboard and system call interfaces)
(4) Using the external library of dynamic string functions.
(5) Using the Random File access routines.
(6) Interfacing machine language programs to Pascal.
(7) Miscellaneous patches to modify TRS-80 Pascal.

4/ TUTORIAL
A step by step introduction to Pascal aimed at people with some knowledge of a computer language.

5/ LANGUAGE REFERENCE MANUAL
A detailed guide to the TRS-80 Pascal language.

6/ ADVANCED DEVELOPMENT PACKAGE
Contains sections on the use and execution of the Native Code-generator and Optimizer programs. Explains when and why to use these utilities.

7/ MASTER CROSS REFERENCE INDEX
A cross reference index for the entire documentation package.

The TRS-80 Pascal system includes the following files:

Disk 1 of 2

PASCAL/CMD	Non-overlayed compiler
ERRORS/DAT	Error message file used by the compiler
RUNP/CMD	Fast load and run utility
TRSLIB/PCL	External declarations for TRS-80 library
STRINGS/PCL	External declarations for STRING library
DATABASE/PCL	Tutorial database program in source form
EDIT/CMD	Text editor
HELP/HLP	Editor help file
CMD/HLP	Editor help file
KEY/HLP	Editor help file
SETEDIT/CMD	Editor setup file utility
SETUP/EDT	Editor binary setup file
SAMPLE/EDT	Sample binary setup file (maps arrow keys)
SYSTEM1/JCL	Configures a minimum system disk #1
SYSTEM2/JCL	Configures a minimum system disk #2

Disk 2 of 2

PASCALB/CMD	Overlayed compiler
PASCAL/OV1	Overlay file for PASCALB
PASCAL/OV2	Overlay file for PASCALB
PASCAL/OV3	Overlay file for PASCALB
PASCAL/OV4	Overlay file for PASCALB
ERRORS/DAT	Error message file used by the compiler
LINKLOAD/CMD	Linking loader utility
TRSLIB/OBJ	Object for TRS-80 runtime library
STRINGS/OBJ	Object for string library
RANDOM/OBJ	Object for Random file procedures
CODEGEN/CMD	Native code generator
CODEINIT/DAT	Data file for CODEGEN/CMD
OPTIMIZE/CMD	P-code Optimizer
SYSTEM3/JCL	Configures a minimum system disk #3
SYSTEM4/JCL	Configures a minimum system disk #4

BEGINNERS GUIDE

Table of Contents

1) Making Backups 2

2) File Configuration 3

3) Overall System View 5

4) Using the Editor..... 6

5) Entering a Simple Program.....12

6) Compiling the Program.....13

7) Running the Program.....15

8) Trouble Shooting Guide.....17

9) Common Error Messages.....18

10) Common Programming Mistakes.....19

MAKING BACKUPS

The first thing you should do before using the TRS-80 Pascal system is to make backup copies of the two supplied master disks. Follow the steps below to create your backup disks.

1) Insert a TRSDOS Version 6 operating system disk into drive 0 and press the reset key.

2) When prompted with Date ? , type in the current date in the form mm/dd/yy and press the <ENTER> key. The screen will then display TRSDOS Ready.

3) Insert a new blank disk into drive 1 and type format :1 <ENTER>. Answer the prompts as follows:

```
Diskette name ? PASCAL1 <ENTER>
Master password ? password <ENTER>
Single or Double density <S,D> ? d <ENTER>
Number of cylinders ? 40 <ENTER>
```

The operating system will now format the disk and display the message "Formatting complete" when finished.

4) Now type: backup :0 :1 (x) <ENTER>
When prompted with Insert SOURCE disk <ENTER>, insert the TRS-80 Pascal disk labeled "Disk 1 of 2" into drive 0 and press the <ENTER> key. The operating system will make a mirror image backup and then display the message: Insert SYSTEM disk <ENTER>. Insert the TRSDOS Version 6 operating system disk back into drive 0 and press the <ENTER> key.

6) Repeat steps 3 and 4 using the diskette name PASCAL2 instead of PASCAL1 in step 3 and "Disk 2 of 2" instead of "Disk 1 of 2" in step 4.

7) Label the backup disks as PASCAL1 and PASCAL2. Place your master TRS-80 disks in a safe place and use the backup copies.

FILE CONFIGURATION

The Pascal system files should now be arranged to provide a useful configuration for program development. How the files are arranged depends on the drive configuration of your Model 4.

Hard Disk Users

If you have a hard disk drive, then one useful configuration is to copy all the TRS-80 Pascal files onto drive 0 of the hard disk. This may be accomplished by using the backup by class command after booting the hard disk system. Place the disk labeled PASCAL1 into one of the floppy drives. If the floppy drive number is 2, then the following command may be used to copy all the PASCAL1 files to drive 0 of the hard disk.

```
TRSDOS Ready
backup :2 :0 (new) <ENTER>
```

Repeat the process using the disk labeled PASCAL2.

Floppy Disk Users

There are many ways of arranging the supplied Pascal files to provide a suitable configuration for program development. How you arrange the files is dependent on the number of drives available.

If you have more than two floppy drives, a useful configuration would be to use drive 0 for the operating system, drive 1 for either the PASCAL1 or PASCAL2 disk, and the remaining drives for storing the programs being developed.

If you have only two drives, follow the configuration steps outlined on the next page. This is a sample 4 disk configuration that combines only the necessary operating system files with selected Pascal files. With this configuration, drive 0 will be used as a system disk (containing the necessary operating system files and selected Pascal files) and drive 1 will be used as a data disk (containing the programs being developed).

Configuration for a 2 drive system

step 1) Make 4 backup copies of your original TRSDOS Version 6 operating system disk and label the backup copies as SYSTEM1 through SYSTEM4.

step 2) Insert SYSTEM1 into drive 0 and PASCAL1 into drive 1.
Type: DO =SYSTEM1 <ENTER>
All unnecessary operating system files will be deleted from SYSTEM1 and the following Pascal files will be copied from PASCAL1 to SYSTEM1: (PASCAL/CMD, EDIT/CMD, RUNP/CMD ERRORS/DAT, SETUP/EDT, SAMPLE/EDT, CMD/HLP, HELP/HLP, KEY/HLP)

step 3) Insert SYSTEM2 into drive 0.
Type: DO =SYSTEM2 <ENTER>
All unnecessary operating system files will be deleted from SYSTEM2 and the following Pascal files will be copied from PASCAL1 to SYSTEM2: (SETEDIT/CMD, STRINGS/PCL, TRSLIB/PCL, DATABASE/PCL)

step 4) Insert SYSTEM3 into drive 0 and PASCAL2 into drive 1.
Type: DO =SYSTEM3 <ENTER>
All unnecessary operating system files will be deleted from SYSTEM3 and the following Pascal files will be copied from PASCAL2 to SYSTEM3: (PASCALB/CMD, PASCAL/OV1, PASCAL/OV2, PASCAL/OV3, PASCAL/OV4, LINKLOAD/CMD, ERRORS/DAT, STRINGS/OBJ, TRSLIB/OBJ, RANDOM/OBJ)

step 5) Insert SYSTEM4 into drive 0.
Type: DO =SYSTEM4 <ENTER>
All unnecessary operating system files will be deleted from SYSTEM4 and the following Pascal files will be copied from PASCAL2 to SYSTEM4: (CODEGEN/CMD, OPTIMIZE/CMD, CODEINIT/DAT)

The disk labeled SYSTEM1 contains all the Pascal files which are necessary to edit, compile, and execute programs. This is the only system disk needed for beginning programmers. The programs on SYSTEM2 through SYSTEM4 are for more advanced programming.

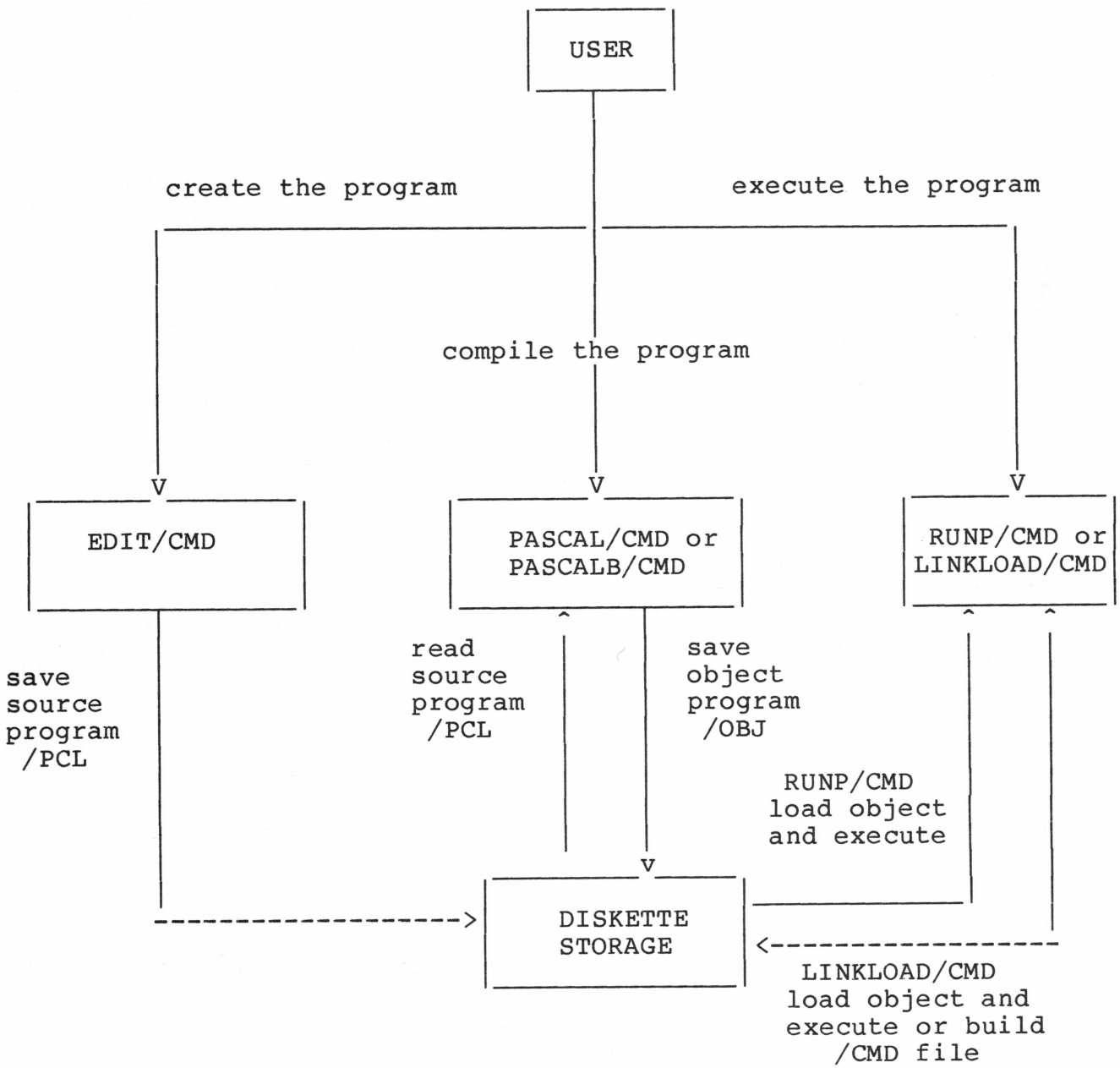
The disk labeled SYSTEM2 contains a utility for creating customized setup files for the editor. A couple of files contain the external declarations for the library routines. A sample Pascal database program is also on this disk.

The disk labeled SYSTEM3 contains the overlayed compiler for compiling large programs, the linking loader for building executable command files, and the object code for the library routines.

The disk labeled SYSTEM4 contains utilities for optimizing the size and/or speed of a program.

Overall System View

The following diagram illustrates the program development process:



The remainder of this manual describes the steps of editing, compiling, and executing a Pascal program. For the following discussion, it is assumed that the files contained on the disk labeled SYSTEM1 (described in File Configuration for floppy disk users) are available. If you have a two drive system, place the SYSTEM1 disk in drive 0. If you have more than two drives or if you have a hard disk, make sure that the files on SYSTEM1 are available on some drive. You should also have a formatted disk with plenty of free space for storing programs. If you have a two drive system, insert a formatted data disk into drive 1.

The editor has many commands which are internally mapped to standard ASCII control codes. These codes are generated from the keyboard by holding down the key labeled CTRL while pressing an alphabetic key. For example, the editor command to move the cursor one character to the right is mapped to CTRL D. (the editor manual contains a complete listing of internally mapped commands). An interesting feature of this editor is that the internal mapping of commands to keys may be changed. This means that you can design the keyboard layout to suit your own personal preferences. As an example, you may want to use the right arrow key (rather than CTRL D) to move the cursor one character to the right.

Each time the editor is executed, it reads a file named SETUP/EDT. The editor uses this file to determine how to operate. At a mimimum, this file must contain information about the Model 4 terminal. For example, the editor must know how to position the cursor on the screen. Optionally, the file may contain information about how editor commands are mapped to keys. For example, it may tell the editor that the right arrow key should cause the cursor to move to the right.

The supplied file named SETUP/EDT contains only information about the Model 4 terminal. This setup file will cause the editor to only understand the internally defined mapping of commands to keys. In other words, the CTRL key is used to execute commands. The supplied file named SAMPLE/EDT is a sample editor setup file that contains the same terminal information but in addition defines a keyboard layout that utilizes the Model 4 arrow keys.

The editor manual describes the operation of the editor based on the internal mapping of commands. At the end of the manual is a sample setup file that defines a keyboard layout that utilizes the Model 4 arrow keys. This is the supplied SAMPLE/EDT setup file. To illustrate how the editor's operation may be altered, this manual will describe the editor commands based on using the SAMPLE/EDT setup file.

Since the editor automatically reads the setup file named SETUP/EDT, you must rename the setup files in order to use SAMPLE/EDT. Type the following commands:

```
TRSDOS Ready
RENAME SETUP/EDT TO SETUP/SAV <ENTER>
RENAME SAMPLE/EDT TO SETUP/EDT <ENTER>
```

Before executing the editor, make sure that there is plenty of disk space for storing files. It is a good practice to write protect the disks which are used as system disks (for example, SYSTEM1) to prevent data from being stored on them. On a two drive system, this will force all files to be stored on the data disk in drive 1.

The editor may be executed from TRSDOS Ready by typing a command of the following form:

```
EDIT filename <ENTER>
```

The filename is optional. If no file is specified, the editor will create a new file. The name of the new file will be specified at the end of the edit session. If a file name is specified, it should be the name of an ASCII formatted text file with record lengths of 80 characters or less. The file name may be any legal TRSDOS file name, including drive specifier. It is suggested that you specify a drive number with the file name. This will cause the editor to place the file on that drive when the editor is exited.

The editor reserves a section of memory which is used as a buffer for storing text. The symbol *EOB is displayed by the editor to indicate the "end of buffer". If no file is specified when the editor is executed, the buffer will start out empty and the *EOB symbol will appear at the top left corner of the screen. If a file is specified, the editor will load in the first 100 lines of the file and display a screen full of lines starting with the first line loaded.

If the buffer is empty, blank lines must be inserted into the buffer before text may be entered. Each time <CLEAR N> is typed (holding the CLEAR key down while pressing the N key), a blank line will be inserted into the text buffer. Once the buffer has lines in it, you may simply type in the text. Typing <LEFT ARROW> will cause the cursor to backspace if you need to correct a typing error. The <ENTER> key will cause the cursor to be positioned to the beginning of the next line. The editor will not allow the cursor to be positioned beyond the *EOB symbol. To enter more text after reaching the end of buffer, type <CLEAR N> to enter more blank lines into the buffer.

The most often used editor commands are the ones which move the cursor around within the text buffer. Most of the cursor movement commands are mapped to the arrow keys.

There are four basic cursor movement commands. (right, left, up, and down). Each of these commands moves the cursor in the specified direction. These commands have been mapped to the arrow keys. They are executed by simply pressing the appropriate arrow key.

Key	Command Name	Function
<RIGHT ARROW>	RT (right)	move cursor right 1 character
<LEFT ARROW>	LF (left)	move cursor left 1 character
<UP ARROW>	UP (up)	move cursor up 1 line
<DOWN ARROW>	DN (down)	move cursor down 1 line

The above commands provide the ability to position the cursor any place on the screen. However, moving only a single character or line at a time can be a little slow. Other commands are mapped to allow you to move the cursor more efficiently.

There are two commands that move the cursor left or right by one tab stop. The tab command moves the cursor to the right to the next tab stop. The back tab command moves the cursor to the left to the next tab stop.

Since the text buffer holds more than a screen full of text, you also need a way to scroll back and forth in the buffer. The roll up command moves the cursor one screen towards the beginning of the buffer while the roll down command moves the cursor one screen towards the end of the buffer.

These commands have also been mapped to the arrow keys. They are executed by holding down the CLEAR key while pressing the appropriate arrow key.

Key	Command Name	Function
<CLEAR RIGHT ARROW>	TB (tab)	move cursor right to the next tab stop
<CLEAR LEFT ARROW>	BT (back tab)	move cursor left to the next tab stop
<CLEAR UP ARROW>	RU (roll up)	move one screen toward the top of the buffer
<CLEAR DOWN ARROW>	DN (roll down)	move one screen toward the bottom of the buffer

Other commands provide the ability to move the cursor greater distances even more efficiently. The beginning of line command positions the cursor at the beginning of the line. The end of line command positions the cursor at the end of the line. The top of buffer command displays the first line in the buffer at the top line of the screen. The bottom of buffer command positions the cursor at the *EOB mark at the end of the buffer.

These commands are also mapped to the arrow keys. They are executed by pressing and releasing the BREAK key and then pressing the appropriate arrow key.

Key	Command Name	Function
<BREAK RIGHT ARROW>	EL (end line)	move cursor to the end of the line
<BREAK LEFT ARROW>	BL (beginning line)	move cursor to the beginning of the line
<BREAK UP ARROW>	TP (top of buffer)	display the first line in the buffer at top of screen
<BREAK DOWN ARROW>	BB (bottom of buffer)	move the cursor to the *EOB mark at the end of buffer

Seven commands are mapped to alphabetic keys. You have already used one, the insert line command. There are three commands that delete either a character, word, or line.

The undelete line command may be used to restore a line that was accidentally removed by the delete line command. There is also a duplicate line command that may be used to make a duplicate copy of the line above the cursor.

The insert character command may be used to insert characters in a line. When this command is executed, subsequent characters that you type will be inserted at the current cursor position. The editor will continue to insert characters until a non-printable character (such as the <ENTER> key) is typed.

These commands are mapped to alphabetic keys. They are executed by holding down the CLEAR key while pressing the appropriate alphabetic key.

Key	Command Name	Function
<CLEAR N>	IL (insert line)	insert a blank line at the cursor line
<CLEAR C>	DC (delete character)	delete character under the cursor
<CLEAR W>	DW (delete word)	delete word under the cursor
<CLEAR L>	DL (delete line)	delete line under the cursor
<CLEAR U>	UL (undelete line)	restore the last deleted line
<CLEAR D>	DU (duplicate line)	duplicate the line above the cursor
<CLEAR I>	IC (insert character)	insert characters until a non-printable is typed

Six other frequently used commands are mapped to the Model 4 special function keys (F1 through F3).

The forward word command moves the cursor to the first character of the word to the right. The backward word command moves the cursor to the first character of the word to the left. Both the forward word and backward word commands will move the cursor across line boundaries.

The split line command creates two lines out of one. This command causes all characters to the right of the cursor to be moved to a new line below. The merge line command is used to merge two lines. As many characters as will fit on a line are moved from the line below the cursor to the end of the line containing the cursor.

The insert mode command is similar to the insert character command. However, it does not terminate when a non-printable character is typed. The editor continues to insert characters until the insert mode command is executed again. This command toggles the editor in and out of insert mode. While in insert mode, the editor inserts a blank line when the <ENTER> key is typed. If the <ENTER> key is typed in the middle of a line, the characters to the right of the cursor are moved to the next line.

The last command that you must know is the command that places the editor in command mode. Command mode allows all editor commands to be executed. This is important since not all commands are mapped to a key. When this command is executed, the editor displays angle brackets <> at the bottom left corner of the screen. Then any editor command may be executed by typing its two character command name followed by the <ENTER> key. For example, UP <ENTER> would execute the cursor up command and then exit command mode.

These commands are executed by pressing the appropriate function key. Three of them require that you hold down the shift key while pressing the function key.

Key	Command Name	Function
<F1>	CM (command mode)	enter command mode
<F2>	BW (backward word)	move the cursor left one word
<F3>	FW (forward word)	move the cursor right one word
<SHIFT F1>	IM (insert mode)	enter permanent insert character mode
<SHIFT F2>	SP (split line)	split the line at the cursor
<SHIFT F3>	MG (merge line)	merge the line below with the cursor line

The commands described so far should be quite adequate for handling most of your editing needs. The editor has many other commands which are described in the editor manual. Once you become familiar with these commands, you will want to read the editor manual for information on other available commands.

For now, the only other commands you must know are the commands to terminate an edit session. To execute these commands, you must enter command mode. As described earlier, pressing the <F1> function key puts the editor in command mode.

Two commands may be used to terminate an edit session. The first command is EX (exit). This command should be used if you wish to save the text to a file. The other command is QT (quit). This command should be used if you wish to terminate the edit session without saving the text.

The EX command requires two parameters, the name of the file to which the text will be written, and whether or not you wish the editor to create a backup file. The editor will prompt you to enter both of these parameters when EX <ENTER> is typed. The first prompt is for the file name. If creating a new file, now is the time that the file name must be specified. Simply type in a valid file name. If editing a pre-existing file, you may simply type <ENTER> to the file name prompt. The text will be written to the file specified when the editor was executed. The second prompt is whether or not to create a backup file. You may answer this prompt by typing either Y for yes or N for no, followed by the <ENTER> key. Simply typing the <ENTER> key for this prompt is equivalent to typing Y <ENTER>. A backup file is created only if the file being edited already exists. The file specified in the EXIT command is renamed with the extension /BAK before the new file is written out. The backup file may be used to restore a file if the file is for some reason damaged. The backup file will reflect one edit session prior to the current one.

The QT command is used to terminate the edit session without saving anything. Simply type QT <enter>. You will then be prompted to make sure that this is what you really want to do. If you answer Y <enter>, the editor terminates. Otherwise, the edit session is continued.

Now that you know how to use the editor, a simple Pascal program may be created. Make sure that the SYSTEM1 disk is in drive 0 and a formatted data disk is in drive 1. Drive 1 will be used to store the program so make sure that there is plenty of free disk space on drive 1 before beginning. First type EDIT <ENTER> and the editor is executed. Since no file was specified, *EOB will appear at the top left corner of the screen. Type <CLEAR N> four times to enter four blank lines into the buffer.

Note: Blank lines are inserted automatically when the <ENTER> key is pressed if the editor is in insert mode. The editor is toggled in and out of insert mode by pressing <SHIFT F1>.

Type in the following text.

```
PROGRAM TEST;
BEGIN
  WRITELN(' THIS IS MY FIRST PROGRAM. )
END.
*EOB
```

Once the text as been entered as shown above, type <F1> to enter command mode. Execute the exit command and answer the two prompts as shown below.

```
<> EX <ENTER>

<EXIT>FILE: TEST/PCL:1 <ENTER>

<EXIT>BACKUP? <ENTER>
```

The program will be saved to the file TEST/PCL on drive 1 and the editor will exit to the operating system. It is important to name your Pascal source files with the extension /PCL because the compiler uses this as the default extension.

COMPILING THE PROGRAM

The compiler must now be used to translate the Pascal program to object format. Once in object format, the program may be executed. Type the following to compile the program created on the previous page.

PASCAL TEST:1 <ENTER>

The Pascal compiler will execute and begin reading the file TEST/PCL on drive 1. As each line is read, it is translated to object code. The compiler will write the object code to the file TEST/OBJ on drive 1. The compiler also sends a listing to the screen as it compiles. The listing will show if there are any errors in the program being compiled. The below listing was generated by compiling the sample test program.

```
TRS80 PASCAL  VER: 00.02.00  14000000  xx:xx:xx xx/xx/xx PAGE 1
-----------------------------------------------------------------
    1|PROGRAM TEST;
    2|BEGIN
    3|  WRITELN(' THIS IS MY FIRST PROGRAM. )
*****                                       ^202
    4|END.
*****   ^ 20,   4,   13
         4 ERRORS DETECTED

  4 ')' EXPECTED
 13 'END' EXPECTED
 20 ',' EXPECTED
202 STRING CONSTANT CANNOT SPAN LINES

STACK USED =  xxxx OF xxxx    HEAP USED = xxxx OF xxxx
```

As you can see, the compiler detected some errors in the program. The compiler always writes an error message line following the line where the error was detected. The error message line begins with 5 asterisks to clearly indicate that an error was detected. It also contains a pointer to the line above at the approximate location of the error. Following the pointer is an error code telling the type of error detected. At the end of the listing, all generated error codes are listed with a brief explanation of the error.

All Pascal programs, including the compiler, use a section of memory which is divided into two parts. One part is the stack and the other part is the heap. The stack is used to store most variables. The heap is used to store dynamic variables and file descriptors. When the compile is finished, the amount of stack and heap used out of the total amount available is displayed on the screen.

Because of the context in a Pascal program, a single error in the program can generate multiple error messages. Usually, the first error code will describe the real cause of the error. In this example, the first error detected is on line 3, error code 202. Error code 202 says that a string constant cannot span lines. This error was caused by the failure to include a closing quote for the string in line 3. The other 3 detected errors are a side effect of the first error. The compiler automatically creates a file named

PASCAL/ERR

when errors are detected in a program. Only the lines containing errors, along with the error message line, are written to this file.

The program should now be corrected before it is executed. The following will cause the editor to execute and display file TEST/PCL on the screen.

EDIT TEST/PCL:1 <ENTER>

Move the cursor to the third line of the program and add a closing quote just prior to the right parenthesis. The third line should then look as follows.

WRITELN(' THIS IS MY FIRST PROGRAM. ')

Type <F1> to enter command mode and type EX <ENTER> to exit the editor. The two prompts may be answered by simply pressing the <ENTER> key.

Now the program may be compiled once again by typing the following.

PASCAL TEST:1 <ENTER>

The following listing will be sent to the screen as the program is compiled.

```
TRS80 PASCAL  VER: 00.02.00  14000000  xx:xx:xx xx/xx/xx   PAGE 1
-----------------------------------------------------------------
   1|PROGRAM TEST;
   2|BEGIN
   3|  WRITELN(' THIS IS MY FIRST PROGRAM.')
   4|END.
TEST
 NO ERRORS DETECTED
NO ERRORS DETECTED

STACK USED = xxxx OF xxxx    HEAP USED = xxxx OF xxxx
```

This time no errors were detected. The program is a legal Pascal program. Now that the program has been compiled with no errors, it may be executed.

Once the program has been compiled without errors, it can be executed with the RUNP command. Type the following to execute the program in file TEST/OBJ on drive 1.

```
RUNP TEST:1 <ENTER>
```

The files used in a Pascal program are logical files. This means that the name of a file used in the program is not necessarily the same name as the actual physical disk file name. When a file is opened within a Pascal program, a prompt will appear on the screen. The prompt identifies the logical file name used in the program. You should then type in the actual disk file name which will be used when the program performs input or output to that logical file. This provides you with the ability to direct input and output to different files or devices each time the program is executed.

Two standard predeclared logical files in Pascal are INPUT and OUTPUT. These files are automatically opened when a program is executed. Therefore, each time you execute a program, the following prompts appear on the screen. (note: the SETACNM library procedure, explained in the System Implementation Manual, may be used to eliminate file prompts)

```
INPUT  =
OUTPUT =
```

The prompts occur for these two logical files whether they are used in the program or not. If you type in a file name, the program will use that file name when performing input or output. You may also simply type the <ENTER> key in reply to a logical file prompt. If the logical file is an input file, input will be received from the keyboard. If the logical file is an output file, output will be sent to the screen (or CRT).

The sample program uses only the logical file OUTPUT. This logical file is implicitly used by the WRITELN procedure. You may simply press the <ENTER> key for both the INPUT and OUTPUT file prompts. The WRITELN procedure will then cause the following to be printed on the screen.

```
THIS IS MY FIRST PROGRAM.
```

When a program terminates, (ie. finishes execution normally), the address of the last instruction executed is displayed on the screen. Following this is the amount of stack and heap used by the program. The stack and heap are explained in the System Implementation manual. The miscellaneous patch section of the System Implementation Manual also explains how to prevent this information from being displayed.

The file names that you type to direct Pascal input and output are the same format as normal TRSDOS file names. The disk drive specification is optional. Devices may also be specified instead of a file name. For example, the name of the line printer is ":L". The name of the terminal which is the keyboard for input and the CRT for output, is ":C". Simply typing the <ENTER> key is equivalent to typing ":C". There is also a dummy device. If a logical file is associated with ":D", then no actual output occurs. This is useful if you wish to discard certain outputs. For example, the listing may be discarded during a compile or you might discard some of the output generated by a program when it is executed.

The Pascal compiler always uses the extension /PCL if the file name is specified when the compiler is executed. The compiler may also be executed by simply typing PASCAL without a file name. If executed in this manner, the compiler will prompt for the Pascal SOURCE file name, the file to use for the LISTING, and the file to use for the OBJECT. Either a file name or a device may be specified. If a file name is specified, the complete file name, including extension, must be used. (ie. the compiler does not use default extensions)

The RUNP utility uses the default extension /OBJ if no extension is specified in the file name. You may also specify an extension if the object code is in a file named with an extension other than /OBJ. For example, RUNP TEST/COD might be used.

Alternate Symbol Representations

The Pascal compiler recognizes alternate representations of certain symbols because not all terminals have the ability to generate them. Either representation may be used.

Symbols with alternate representations:

symbol	generated on Model 4 by	alternate
{	clear shift <	(*
}	clear shift >	*)
^	clear ;	@
[	clear <	(.
]	clear >	.)

(Miscellaneous Errors)

1. Problem - While editing a file, the latter part of a file is found to be missing.

   Answer - Need to use the APPEND command to page the latter part of the file into the text buffer. See the Editor Manual.

2. Problem - Upon exiting the editor, a PHYSICAL IO error message is displayed.

   Answer - The disk is full. Make sure that there is plenty of free disk space when editing files.

3. Problem - During a compile, the Pascal compiler abnormally terminates with a FATAL ERROR - OUT OF HEAP, or OUT OF STACK

   Answer - The compiler does not have enough memory. Use the PASCALB version of the compiler.

4. Problem - When executing your compiled program with the RUNP command, or a command (/CMD) file built with the LINKLOAD utility, it abnormally terminates with the FATAL ERROR - OUT OF HEAP, or OUT OF STACK

   Answer - Specify the amount of stack when using the RUNP command or the build command of the linking loader. (See the System Implementation Manual)

5. Problem - After executing the compiler using the long form where the OBJECT and LISTING files are specified, the original source file suddenly contains object code.

   Answer - The /PCL extension was used when specifying the object file.

COMMON ERROR MESSAGES

(By the Compiler)

13 END expected - There must be an END for every BEGIN in a Pascal program.

52 THEN expected - IF statements require use of the reserved word THEN.

54 DO expected - FOR statements require use of the keyword DO.

104 undeclared identifier - All variables must be declared in a Pascal program.

119 semicolon expected - The preceding declaration or statement is not terminated by a semicolon (;).

127 type of actual parameter does not match formal parameter - An attempt to call a procedure or function with an argument that does not match the type of the formal parameter. In special cases, the type matching requirements may be overridden by using the type transfer operator (::).

129 type conflict of operands in an expression - An attempt to use an operator with two variables of different types.

154 actual parameter must be a variable - Using a constant instead of a variable when calling a procedure whose formal parameter is preceded by VAR.

--- Unexpected End of File - A period does not follow the last END of the program or a comment is missing the closing comment symbol } or *).

(Runtime Error Messages)

The error codes discussed above are generated by the compiler due to an error in the Pascal source program. There are times when the compiler may generate a fatal error message that is not due to an error in the source program. These are called runtime errors because they are detected by the runtime that is included with all Pascal programs, including the compiler. The following are examples of runtime errors.

RUNTIME ERROR 01 OUT OF STACK
(Caused by trying to compile or run too large of a program)
RUNTIME ERROR 02 OUT OF HEAP
(Caused by trying too compile or run too large of a program)
RUNTIME ERROR 09
(file not found or disk error)

Note: Explanation of COMPILER and RUNTIME error codes may be found in the appendix of the Reference Manual.

1. For variable types to match in expressions and not generate compiler error messages, they must be explicitly declared to be of the same type in the declaration section. For example:

```
PROGRAM TEST;
VAR A:ARRAY[1..5]OF CHAR;
    B:ARRAY[1..5]OF CHAR;
BEGIN
   A:=B;
END.
```

will generate a type conflict message by the compiler although the types appear to match. The following examples will not generate an error message and are perfectly legal in Pascal.

```
PROGRAM TEST;
VAR A,B:ARRAY[1..5]OF CHAR;
BEGIN
   A:=B;
END.
```

```
PROGRAM TEST;
TYPE D = ARRAY[1..5]OF CHAR;
VAR A  :D;
    B  :D;
BEGIN
   A:=B;
END.
```

2. A procedure declaration that has non-NAMED types in the parameter list is illegal in Pascal. (ie. the following is illegal.)

```
PROGRAM TEST;
  PROCEDURE EXAMPLE(VAR A:ARRAY[1..5] OF CHAR); EXTERNAL;
BEGIN
END.
```

The following is legal:

```
PROGRAM TEST;
TYPE D = ARRAY[1..5] OF CHAR;
  PROCEDURE EXAMPLE(VAR A:D); EXTERNAL;
BEGIN
END.
```

3. Placing a ; (semicolon) before the ELSE part of an IF THEN ELSE statement is illegal.

```
illegal:    IF X=Y THEN A; ELSE B

  legal:    IF X=Y THEN A  ELSE B
```

EDITOR MANUAL

Table of Contents

1) INTRODUCTION..... 3

2) BLAISE II OVERVIEW..... 4

A. Editor Setup Files..... 4
B. The SETEDIT Program..... 5
C. Text Buffer Management..... 6
D. The Work File..... 7
E. Compose Mode..... 7
F. Command Mode..... 8

3) GETTING STARTED..... 9

A. Editor File Configuration..... 9
B. Terminal Configuration.....10
C. The EDIT Command.....12
D. Basic Editor Commands (keyboard mapped).....13
E. Editor Help Files.....18
F. Sample Edit Session.....19
G. Swapping Disks During Edit Session.....21

4) EDITOR COMMANDS.....22

A. Command Parameters.....22
B. Editor State Commands.....24
1. No Parameters.....24
2. Parameters.....26
C. Cursor Positioning Commands.....27
1. No Parameters.....27
2. Parameters.....28
D. Character Commands.....30
E. Line Commands.....31
F. String Commands.....32
1. No Parameters.....32
2. Parameters.....32
G. Block Commands.....34
1. No Parameters.....34
2. Parameters.....35
H. File Commands.....36
I. General Commands.....41
J. The EDIT Command.....42

5) CHANGING EDITOR CHARACTERISTICS.....43

A. Translating Keyboard Characters to Commands.....43
B. Defining Macro Commands.....45

6) EDITOR SETUP FILES.....48

A. Normal Commands.....49
1. The TABS Command.....49
2. The ROLL Command.....49
3. The AUTOINDENT Command.....49
4. The TRANS Command.....49
5. The DEFINE Command.....49

B. Special Commands.....50
1. The INIT Command.....50
2. The EXIT Command.....50
3. The START Command.....50
4. The CMD Command.....51
5. The HEIGHT Command.....51
6. The WIDTH Command.....51
7. The TERMINAL Command.....52
8. The CURSOR Command.....53

C. Sample Setup file.....54

APPENDIX

A. Sample Custom Terminal Setup.....57

INTRODUCTION

A text editor is simply a program that is used to enter text and save the text to a file. Usually, text editors are classified into one of two categories, line or screen editors.

Line editors are called as such because they operate on text a line at a time. Typically, you must view the text being edited by listing the lines that fall between two specified line numbers. Usually, the text must be modified by typing commands which operate on a specified line number. To change a character on a line, you often must use a command rather than being able to position the cursor to the bad character and correcting it.

Screen editors are called as such because they operate on a screen full of text at a time. A screen editor gives you much more context when editing a file. You are able to move the cursor around on the screen, changing the text by simply typing over the incorrect text. Rather than thinking in terms of line numbers, a screen editor allows you to scroll through the text a page at a time. A screen editor makes editing much easier by providing more powerful commands and an environment which allows you to see what you are changing as you change it.

The Blaise II text editor is a screen editor. It provides a very good tool for entering your programs or other textual documents. Some of the features found in word processors have been included in the Blaise II editor. Although it was designed for program entry, you may find that it serves many of your word processing needs as well.

This chapter provides a brief description of the major features of the Blaise II text editor. It should provide you with a basic understanding of how the editor operates. The actual use of the editor is more completely explained in the following chapters.

Blaise II is a very powerful text editor with many commands and features. It is designed so that the user may change the characteristics of the editor to conform to personal preferences. As supplied, the editor is internally configured to map the most frequently used commands to the keyboard. These commands are executed by typing control characters. It is suggested that the inexperienced user learn how to use the editor with this standard configuration. The experienced user may want to alter the editor characteristics so that it operates similar to some other familiar editor.

A. Editor Setup Files

Before the editor may be used, it must be configured for the type of terminal used by your computer. The editor uses what is called a setup file to perform this configuration. A setup file is simply a file containing commands that the editor understands. Each time the editor is executed, it loads in the setup file and configures itself based on the these commands. The commands in a setup file tell the editor about the terminal's special features.

The setup file may also contain commands that cause the editor to default to some desired state. You may do things such as define how keys are mapped to editor commands, or define new commands which are composed from the set of built in commands.

B. The SETEDIT Program

The SETEDIT program is a utility program used to create setup files for use with the editor. This utility provides a menu of commands which allow various setup file related operations to be performed. Following is a list of the commands.

```
T = define terminal characteristics
R = read a text format setup file
W = write a binary setup file
I = input a binary setup file
O = output a text setup file
H = display help information
E = exit
```

The SETEDIT utility must at a minimum be used to create a setup file which contains terminal information. The editor must know what kind of terminal is being used before it can properly display text. The T command provides a list of commonly used terminals. If your system uses one of the listed terminals, then terminal definition is accomplished simply by selecting the proper terminal.

The editor requires binary formatted setup files. Once the proper terminal has been selected, a binary setup file may be created using the W command. The editor uses the file SETUP/EDT as the default setup file. When the editor is executed, it loads this file if no setup file is specified. Once the terminal information is written to this setup file, the editor may be used.

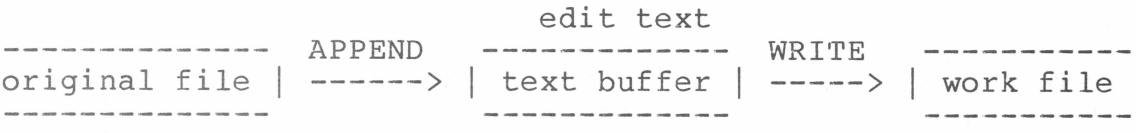
If you wish to view or modify the terminal information created by the SETEDIT utility, you may use the O command to output the information in text (readable) form. It is important to note that the editor can edit text formatted files only. However, the setup files used to configure the editor must be in binary format.

The SETEDIT utility has the ability to read either text or binary formatted setup files with the R and I commands respectively. A binary formatted setup file must have been previously created by SETEDIT. A text formatted file may be created using the editor. The SETEDIT utility may be used to combine multiple setup files and terminal information into one single setup file. For example, after creating the binary setup file containing the terminal description only, the editor may be used to create a text formatted setup file with other commands. Then the SETEDIT utility may be used to read both the files and write the combined information out to another file.

C. Text Buffer Management

The editor maintains a fixed size buffer for storing text. The buffer will hold approximately 10500 characters on the Model 4. All editor commands except for specific file commands operate only on the text in this buffer. When editing very large files, the file must be edited a section at a time. Starting at the beginning of the file, a section is loaded into the text buffer. Before loading another section of the file into the buffer, buffer space must be made available by writing the text out to a work file. Then the next section may be loaded into the buffer. This process may be repeated until the whole file has been loaded and edited.

When editing an existing file, the editor loads the first 100 lines only. This leaves ample buffer space for adding more lines and performing the various editing functions. If the file is longer than 100 lines, the APPEND command may be used to load more text from the file into the buffer. With this command, you specify how many lines to copy from the file to the buffer. The copying begins one line past the last line previously loaded from the file. The text being copied from the file is appended to the end of the text in the buffer. If the file is very large, it is possible for the buffer to become full. If this happens, a MEMORY EXHAUSTED message is displayed. The WRITE command must then be used to write some of the text in the buffer back out to a work file. With this command, you specify how many lines to copy from the buffer to the work file. The copying begins with the first line in the buffer and continues until either the buffer is empty or the specified number of lines have been written. Once lines have been written from the buffer to the work file, they may not be edited again during the current edit session. The following diagram illustrates this process.



If the editor is exited before the entire file has been loaded into the buffer, the editor will copy the remaining lines in the original file to the work file.

The editor displays one of two symbols at the end of the text buffer. The EOB symbol signifies the end of the buffer. If editing a pre-existing file, the symbol displayed will be EOF if all text has been loaded from the original file into buffer.

D. The Work File

The editor creates temporary files during an edit session. These temporary files are called work files. The main work file is a copy of the file being edited. Its purpose is to prevent a system crash from damaging the original file. If such a crash occurs during an edit session, the original file is left unchanged. The main work file is named T0n1.TMP where n is either 1 or 2 depending on the level of the edit. The level is 1 when editing a single file. If the EDIT command is used during an edit session, a second work file is created, level 2. After successfully exiting, if the work file is on the same disk as the original file, then the work file is renamed as the original and the original file is either deleted or renamed as a backup file. If the work file is on a separate disk from the original, then it is copied over the original after which the work file is deleted.

The work file is usually placed on the same drive as the original file. If you are creating a new file and the WRITE command is used, the work file is placed on the default drive. When the work file is on the same drive as the original, there must be enough free disk space to accomodate two copies of the original file.

The editor block movement commands also cause the editor to create a work file. This work file is used as temporary storage for the block of data being moved. The name for this work file is T0n3.TMP where n is either 1 or 2 as above. The editor does not delete this work file.

E. Compose Mode

The editor has two modes of operation, compose mode and command mode. When the editor is executed, it starts out in compose mode. In this mode, you may type in text much the same way as you would with a typewriter. The text that you type is stored in the text buffer. While in this mode, you have access to many editor commands. For example, there are commands to move the cursor around on the screen. These commands are mapped to specific control characters which may be generated from the keyboard. A control character is generated by holding down the key labeled CTRL while pressing an alphabetic key. For example, CTRL X will cause the cursor to move down one line, CTRL G deletes the character under the cursor, etc. Therefore, compose mode allows you to enter text and to move around in the text performing various operations with control characters.

F. Command Mode

One control character is mapped to a command which causes the editor to switch from compose mode to command mode. When CTRL Z is typed, the editor displays angle brackets <> at the bottom left of the screen and goes into command mode.

Command mode provides the ability to execute any command in the editor. Since not all editor commands are mapped to control keys, it is necessary to go into command mode to execute the unmapped commands. All commands in the editor, whether mapped to keys or not, have a two character mnemonic. While in command mode, the commands are executed by typing the two character mnemonic followed by the <enter> key.

Command mode provides a convenient alternative method of executing editor commands. The editor has so many different commands that it is impossible to map all commands to keys in a manner that is logical and easily remembered. A mnemonic is often easier to remember than a control character sequence. The two character mnemonic reflects the actual function of a particular command while a control character sequence may not.

Command mode is entered from compose mode by typing CTRL Z. Angle brackets appear at the bottom left corner of the screen and the cursor is placed to the right of the brackets. Any command may then be executed by typing its two character mnemonic followed by the <enter> key. If there are typing errors, CTRL H may be used to backspace and make corrections. Once the command has finished execution, the editor returns to compose mode. The screen will reflect any changes caused by the execution of the command. To return to command mode, a CTRL Z must be typed once again.

A convenient way of operating the editor is to execute often used commands from compose mode and seldom used commands from command mode. For example, the cursor positioning commands (cursor right, cursor left, cursor up, cursor down) are used constantly. It would be inconvenient to execute these commands from command mode. On the other hand, the directory command (DI) is seldom used. Rather than try to remember what control sequence it is mapped to, it may be easier to remember the mnemonic DI and execute this command from command mode.

The text editor is composed of the main command file EDIT and several help files with the extension HLP. The help files contain information about the editor commands and may be accessed during an edit session. They are not necessary for the operation of the editor. They simply provide helpful information if needed.

The SETEDIT command file is a utility for creating a setup file which the editor must use to obtain information about the terminal of the computer system. Before the editor may be used, the SETEDIT utility must first be used to create a binary setup file named SETUP (with the extension EDT). Each time the editor is executed, it loads this file and configures itself by executing the commands in the setup file. Since the terminal is an integral part of some computer systems, it is possible to configure a setup file for these systems and include it on the master disk. If your disk contains a file named SETUP, then it is not necessary to execute the SETEDIT utility.

A. Editor File Configuration

The EDIT command file and the SETUP default editor setup file are both necessary to execute the editor. The help files (HLP extensions) are not necessary unless you wish to use them to display information about the editor during an edit session. The files may be placed on any drive number.

B. Terminal Configuration

A setup file called SETUP (with extension EDT) must be created before the editor may be used. On some systems, this file may already exist. If so, then this section may be skipped.

The SETEDIT utility must be used to define the characteristics of your particular terminal. The SETEDIT utility contains built in tables for the most widely used terminals. If you are using one of these terminals, then defining the terminal characteristics is a simple matter of selecting the proper terminal from a menu.

The following steps will guide you through the creation of the editor setup file.

1) Type SETEDIT <enter> to execute the utility

2) A menu will be displayed followed by a prompt to enter a selection. Type T <enter> to define terminal characteristics.

3) A list of terminals will be displayed, each preceded by a number. Type <enter> to view the remainder of the built-in terminal types. You will then be prompted to select a terminal. If your terminal is listed, then type in the correct number and proceed to step 5.

   Otherwise, you should select CUSTOM.

   Once CUSTOM has been selected, you will be prompted for the type of cursor adressing. You must specify either binary or ascii. The cursor is positioned to a particular location on the screen by specifying a row and column number. Some terminals use a single character to specify the row or column number. This is binary addressing. Other terminals expect a sequence of ascii digits to specify the row or column. This is ascii addressing. Normally, the 0 row or column position is not addressed with a 0 value. The next prompt will ask for an offset value corresponding to row or column 0.

The next prompt is for which comes first, the row or column address? You will then be prompted for the character sequence preceding the row/column address. Next come two prompts for the character sequence between the row/column addresses and following the row/column addresses. If your terminal does not require any such sequence, simply type <enter> to these two prompts. Proceed to step 4.

4) You will be prompted for information about your terminal's characteristics and will need your terminal manual to answer the questions. The first two prompts are for the HEIGHT and WIDTH of your terminal. The height is the number of lines on the screen. The width is the number of characters on a line. The next sequence of prompts ask "does your terminal have this function?". If you answer yes, then you will be prompted to enter the character sequence to perform that particular function. (see the example in the appendix) The following is a list of the functions which the editor supports.

clear to end of screen	- clear the screen from the current cursor position to the end of the screen.
clear to end of line	- clear the line from the current cursor position to the end of line.
insert line	- insert a blank line at the current cursor positon.
delete line	- delete the line at the current cursor positon.
delete character	- delete the character at the current cursor position.
enter insert mode	- cause the terminal to insert all subsequent characters at the current cursor position.
exit insert mode	- cause the terminal to stop inserting.
scroll 1 line down	- shift each line on the screen down by one line.
insert 1 character	- insert a character at the current cursor position.
scroll 1 line up	- shift each line on the screen up by one line.

5) The main menu is now displayed. Type W <enter> to write a binary setup file and you will be prompted for a file name. Type the name SETUP with the extension EDT and press the <enter> key. The file will be written to disk. A drive specifier may also be included as part of the file name.

6) The main menu is once again displayed. Type E <enter> to exit the program. The setup file has been created and the editor may now be used.

C. The EDIT Command

The editor may be executed in several different ways. When creating a new file, simply type EDIT <enter>. The editor will configure itself with the default setup file and display *EOB at the top left corner of the screen indicating an empty text buffer. You may then insert one or more blank lines and start entering text.

The second way of executing the editor is to type EDIT FILENAME, where FILENAME is the name of some pre-existing text file. A drive specifier may also be included in the file name. The editor will configure itself with the default setup file and then load the first 100 lines from FILENAME. The message LOADING... will be displayed at the bottom of the screen. Once the first 100 lines have been loaded, the editor will display a screen full of text starting with the first line and position the cursor at the top left corner of the screen. You may then begin editing.

The third way of executing the editor is to specify two file names in the form EDIT FILENAME1 FILENAME2, where FILENAME1 is the name of the file to be edited and FILENAME2 is the name of a setup file for the editor to use in configuration. As noted earlier, the editor by default loads the setup file named SETUP (with the extension EDT).

By specifying the setup file on the command line, the editor may be forced to use a setup file of some other name. This is convenient if you want the editor to default to different states, depending on the type of editing being performed. Drive specifiers may be included in either file name.

D. Basic Editor Commands

The complete set of editor commands is explained in the following chapter. This section explains how a selected subset of the commands is mapped to the keyboard.

The editor has a very large set of commands. Some of the commands will be frequently used while others will be used with much less frequency. The commands which are used most often have been mapped to the keyboard. These commands may be executed while in compose mode by typing control characters. The remainder of the commands must be executed by entering command mode and typing a two character mnemonic.

Control characters are generated by holding down the CTRL key while pressing another key. On the Model 4, the editor commands have also been mapped so that the CLEAR key may be used in place of the CTRL key if desired.

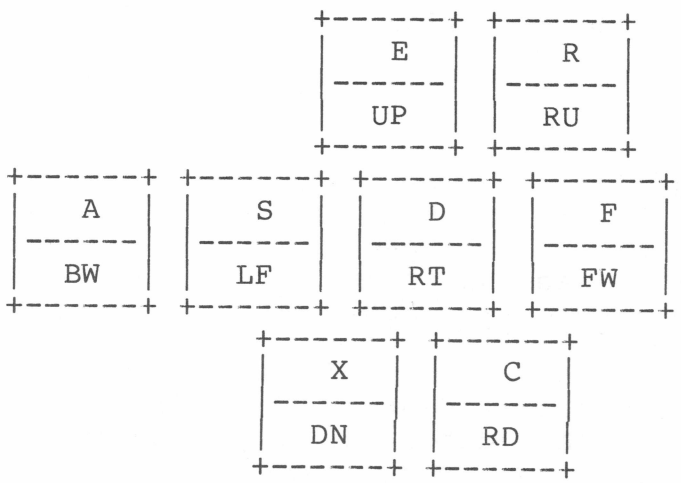
Command mode is entered by typing CTRL Z. Command mode allows you to execute one command and then the editor returns to compose mode. While in command mode, typing CTRL Z before executing a command will also return the editor to compose mode.

When creating a new file, the editor starts out with an empty text buffer. The symbol *EOB will appear at the upper left corner of the screen. Before entering text, one or more blank lines must be inserted into the buffer. The insert line command has been mapped to CTRL N. Each time CTRL N is typed, a blank line will be inserted into the buffer. Once the buffer contains lines, you may begin entering text. The <enter> key will cause the cursor to go to the beginning of the next line.

The editor defaults to overwrite mode. In overwrite mode, the editor will write directly over the character under the cursor. If there is text to the right of the cursor, the text will be changed as you type. The other mode is insert mode. There are two commands that cause the editor to enter insert mode. When CTRL V is typed, the editor temporarily enters insert mode. When characters are typed, they are inserted at the current cursor position. All characters to the right of the cursor will shift one character to the right each time a character is typed. When any editor command is executed, such as <enter>, the editor goes back into overwrite mode. The editor may be permanently placed in insert mode by entering command mode and typing IM <enter>. It may be placed back in overwrite mode by entering command mode and typing IM <enter> once again. This command toggles the editor from overwrite mode to insert mode and vice versa.

When the editor is in permanent insert mode, the <enter> key will insert carriage control. If the <enter> key is typed at the end of a line of text, the editor will insert a blank line following the current line and place the cursor at the beginning of the blank line. If the <enter> key is typed while in the middle of a line of text, the line will be split with the characters to the right of the cursor being placed on the inserted line. Insert mode is most useful when creating new text. It prevents the need for inserting blank lines in the text buffer before entering the text. When the buffer is empty, the <enter> key may be typed to insert one blank line and a new line is inserted automatically each time <enter> is typed thereafter.

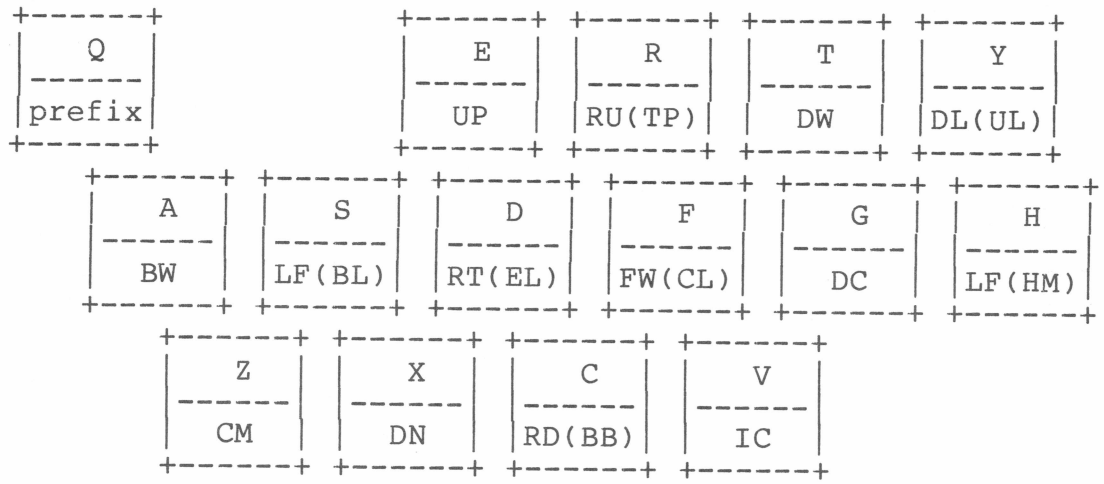
The most frequently used editor commands are the cursor movement commands. These commands have been positionally mapped on the left side of the keyboard. The basic cursor movement commands are cursor right (RT), cursor left (LF), cursor up (UP), and cursor down (DN). These commands are mapped to the D,S,E, and X keys respectively. For example, typing CTRL D will cause the cursor to move one character to the right. Moving the cursor by word is a frequently used command. The (FW) command will move the cursor forward one word, while the (BW) command moves the cursor back one word. These commands have been mapped to the F and A keys respectively. The roll up command (RU) will scroll the text toward the beginning of the text buffer while the roll down command (RD) scrolls the text toward the end of the buffer. The number of lines scrolled is defaulted to 3 lines less than the size of the screen. These commands have been mapped to the R and C keys respectively. The following diagram illustrates the positional mapping used for these commands.



As noted earlier, command mode (CM) is mapped to Z and insert character mode (IC) is mapped to V. The delete character command (DC), which deletes the character under the cursor, is mapped to G. The delete word command (DW), which deletes the current word under the cursor, is mapped to T. The delete line command (DL), which deletes the current line under the cursor, is mapped to Y. Since CTRL H is typically associated with back space, it has been mapped to perform the cursor left command, same as CTRL S.

In addition to the single control character mapping, some commands have been mapped to a two character sequence. CTRL Q is used as a prefix for the commands labeled inside parentheses in the diagram below. After typing CTRL Q, these commands may be executed by typing the single character alone, or by typing CTRL <character>. For example, you could type CTRL Q, R or CTRL Q, CTRL R.

The top of buffer command (TP) causes the screen window to be moved to the top of the text buffer. The bottom of buffer command (BB) causes the screen window to be moved to the bottom of the text buffer. The beginning of line command (BL) causes the cursor to move to the leftmost column on the screen. The end of line command (EL) causes the cursor to move one character past the rightmost character on the line. The center line command (CL) causes the current line to be centered on the screen. The home command (HM) causes the cursor to be positioned at the top left corner of the screen. The editor maintains a buffer for storing the last deleted line. The undelete line command (UL) will insert the last deleted line at the current cursor position.

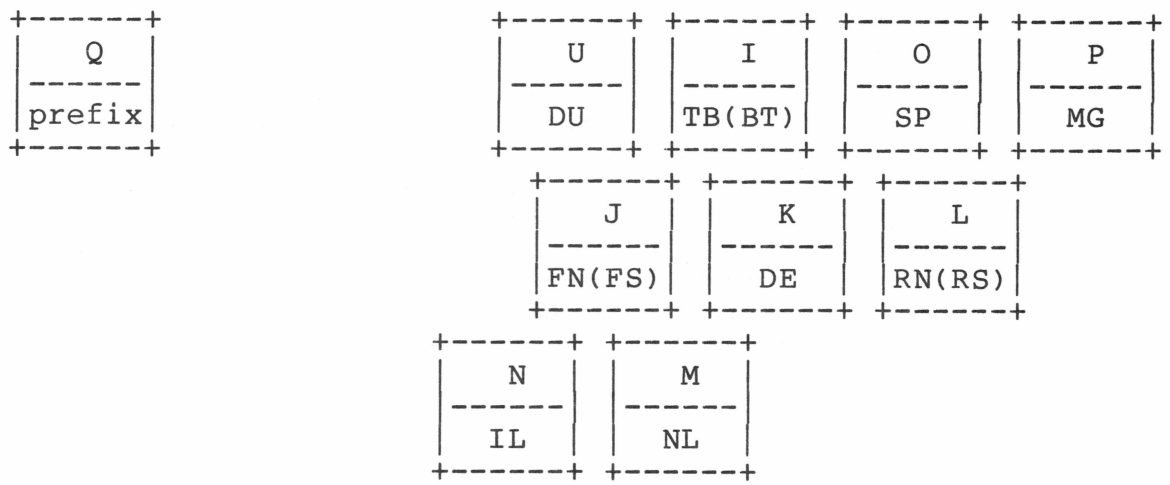


Other frequently used commands have been mapped to the right side of the keyboard. Since CTRL I generates the same character as the tab key, it has been mapped to the tab command (TB). Tabs are defaulted to 4 spaces. The back tab command (BT) has also been mapped to I. Back tab is executed when the CTRL Q prefix is used.(ie. CTRL Q, I or CTRL Q, CTRL I).

The find next string command (FN) which searches for the next occurrence of a string has been mapped to J. Before executing FN, the find string command (FS), which defines a string and then searches for it, should be used. The FS command has been mapped to J with the Q prefix. The replace next string command (RN), which replaces the next occurence of one string with another, has been mapped to L. Before executing RN, the replace string command (RS), which defines a string to search for and another string to use as replacement, should be used. The RS command has been mapped to L with the Q prefix.

The split line command (SP), which causes the current line to be split into two lines at the cursor position, has been mapped to O. The merge line command (MG), which causes the line following the cursor line to be merged with the cursor line, has been mapped to P. The delete to end of line command (DE), which deletes the current line from the cursor to the end of the line, has been mapped to K. The duplicate line command (DU), which duplicates the line above the cursor starting at the current cursor column, has been mapped to U.

As noted earlier, the insert line command (IL) has been mapped to N. CTRL M generates the same character as <enter> and is therefore equivalent. The mnemonic used for <enter> is (NL) which stands for next line.



There are several commands related to block operations. These commands have been mapped mnemonically, rather than positionally. (ie. the commands are mapped to keys that correspond to the first letter of the command mnemonic) The block commands must be prefixed by CTRL B. Simply type CTRL B, CTRL <key> or CTRL B, <key> where <key> is the key to which the specific command has been mapped. For example, CTRL B, M executes the mark command.

Any block operation must have a defined block of text to which the operation is applied. The mark command (MK) places an invisible mark at the line containing the cursor. Any block operation will be applied to the block of text between the marked line and the line currently containing the cursor. All block operations, except UR and LR occur on line boundaries. (ie. the column position of the cursor has no effect)

The three fundamental block operations are copy block (CB), insert block (IB), and delete block (DB). The copy block command copies the text within the marked region (ie. the text between the marked line and the line containing the cursor) to a temporary file. The insert block command inserts the text in the temporary file into the text buffer prior to the the line containing the cursor. The delete block command deletes the text within the marked region.

The print block command (PR) prints the text in the marked region. A line printer must be connected to use this command. The fill command (FI) requires two parameters, the left and right margins (column numbers). The fill command rearranges the text within the marked region so that it fits within the specified margins. The justify command (JF) also requires the same two parameters. The justify command rearranges the text on each line within the marked region so that the text aligns at both margins. This will cause extra blanks to be placed between words.

The upper case and lower case commands are the only block commands that operate on character boundaries rather than line boundaries. The upper case command (UR) makes all characters within the marked region upper case characters. The lower case command (LR) makes all characters within the marked region lower case characters.

There are two other very useful commands associated with the marked line and the line containing the cursor. The swap command (SW) swaps the marked line and the line containing the cursor. The cursor line becomes the new marked line and the cursor is positioned to the previously marked line. The go to mark command (GM) causes the cursor to be positioned to the marked line.

E. Editor Help Files

The files named with HLP extensions are editor help files. These files contain information on editor commands. There are help files on the following subjects: (HELP, KEY, and CMD).

HELP displays information about the other two help files. KEY shows how the commands are mapped to keys. CMD lists all the editor commands in alphabetic order.

The help files may be viewed by typing CTRL Z to enter command mode and then typing HELP <enter>. When prompted for the subject, type in one of the above listed subjects followed by the <enter> key. If the <enter> key is typed without specifying a subject, the HELP file will be displayed. Be sure that all the help files are placed on the system drive.

When viewing a help file, you may scroll downward towards the end of the file by typing CTRL C. To scroll back towards the beginning of the file, type CTRL R. To resume the edit session, type CTRL Z.

F. Sample Edit Session

The following steps show how a new file is created using the editor. Then the editor is used to edit the previously created file. Make sure the editor setup file and help files are on the system drive before beginning.

1) Type EDIT<enter>

2) When *EOB appears at the top left corner of the screen, type CTRL N several times. Each time you type CTRL N, a blank line is inserted into the text buffer and the *EOB symbol moves down one line.

3) Now simply type in the text. When you type the <enter> key the cursor will go to the beginning of the next line. The cursor will not move beyond the *EOB symbol. If you wish to type in more lines, use CTRL N to insert more blank lines into the buffer.

4) Several commands may be used to move around in the text to make changes or corrections. The previous section explained the commands which are mapped to keys. These commands are executed by holding down the CTRL key while pressing one of the alphabetic keys. For example, CTRL S moves the cursor left one character. Some of the commands must be prefixed by CTRL Q. For example, typing CTRL Q and then the S character will cause the cursor to go to the beginning of the current line.

5) If you forget how the commands are mapped to keys, type CTRL Z. Angle brackets will appear at the bottom left corner of the screen. Type HELP KEY <enter>. The screen will display help information about how the commands are mapped to keys. Type CTRL C to move forward in the help file. Type CTRL R to move backward. Type CTRL Z to resume the edit session.

6) Once you have finished entering the text, type CTRL Z to enter command mode. Type EXIT <enter>. You will be prompted with <EXIT>FILE:. Type in a valid file name. Drive numbers may be used as part of the file name. You will then be prompted with <EXIT>BACKUP? Simply press the <enter> key. Your file will be saved and the editor will exit back to the operating system.

7) If you wish to modify the file, type EDIT followed by the filename used in step 6. The editor will load in the first 100 lines of the file. If fewer lines than this were in the file, the whole file will be loaded. Use the editor commands to move around in the text, making modifications. If not all lines were loaded into the text buffer, type CTRL Z to enter command mode and type APPEND 100 <enter>. One hundred more lines will be loaded from the file into the text buffer. If the buffer becomes full, (indicated by OUT OF MEMORY message at the bottom of the screen) type CTRL Z to enter command mode. Type WRITE 100 <enter> and the first 100 lines in the buffer will be written to the editor work file. The lines written may not be edited again until the current edit session is terminated. After freeing buffer space with the WRITE command, more lines may be appended into the text buffer. When finished making changes, type CTRL Z to enter command mode.

8) If you wish to save your changes type EXIT <enter>. You may simply press the <enter> key to answer the following two prompts, <EXIT>FILE: and <EXIT>BACKUP?. The editor will will save your changes to the file specifed in step 7. However, before doing so, it will rename the original file created in step 6 to become a backup file. The same prefix of the file name is used with the extension BAK to represent that it is a backup file. The editor by default creates this backup file. To prevent its creation, you must type N <enter> to the <EXIT>BACKUP? prompt.

9) If you wish not to save your changes, type QUIT <enter>. When prompted with <QUIT>REALLY?, type Y <enter>. The editor will exit to the operating system and the file created in step 6 will be left unchanged.

G. Swapping Disks During an Edit Session

Sometime during an edit session, you may need to access files which are not on any of the disks currently in the drives. It is possible to swap disks during an edit session for such situations. When swapping disks, the swap disk editor command (SD) must be executed each time the disks are swapped. Type CTRL Z to enter command mode and then type SD <enter>.

When the editor is executed and has finished loading the setup file, the EDIT command file and the setup file are no longer needed during an edit session. The other files involved include the original file being edited and the editor work files. These files may be swapped as long as the following rules are followed.

1) The original file must be swapped back before an APPEND operation.

2) The main work file must be swapped back before a WRITE operation.

3) The block command work file must be swapped back before a block operation.

4) The original file and main work file must be swapped back before an EXIT or SAVE operation.

The previous chapter explained only those commands which are internally mapped to the keyboard. The mapped commands may be executed by typing the appropriate control characters. This chapter explains all the editor commands except for a few special setup file commands. All these commands may be executed from command mode.

CTRL Z causes the editor to enter command mode. While in command mode, CTRL H may be used to backspace and correct typing errors . Commands are executed when the <enter> key is pressed. CTRL Z may be used prior to pressing the <enter> key to abort command mode and reenter compose mode.

A. Command Parameters

All commands, some of which require parameters, have an associated two character mnemonic. In addition, the commands which require parameters have command names which may alternatively be used in place of the mnemonic. For example, the find string command requires a string parameter. The find string command may be executed from command mode by typing either FS or FIND, followed by the <enter> key. Abbreviations of the long form names are also accepted. (eg. FIN will work)

When executing commands which require parameters, you may specify the parameters after the command name or you may simply type the command name without specifying the parameters. If the parameters are not specified, the editor will prompt for the required parameters. This is the case for key mapped commands which require parameters (eg. the FIND string command). The prompt will contain the long form of the command name inside angle brackets, followed by the parameter being requested. For example, typing FS <enter> while in command mode will result in the prompt <FIND>STRING:. You must then type in the string.

There are three types of parameters that are used for commands. Integer parameters are required for commands such as FILL (FI) and JUSTIFY (JF). With these commands, you must specify the columns to use as the left and right margins. For example, FILL 10 70 and JUSTIFY 5 75 might be used. When specifying integer parameters, blanks must be used to separate the individual parameters. Some commands have parameters which require a yes/no answer. These may be answered by typing YES or NO or by simply typing Y or N. Both upper and lower case are accepted. The third type of parameter is a string. There are multiple ways of specifying string parameters. They may be quoted using either single or double quotes, or they may be unquoted. For example, either FIND 'ABC' or FIND "ABC" or FIND ABC could be used to locate the string ABC.

When specifying multiple string parameters on the command line, all but the last string parameter must be quoted. If the editor is expecting a string parameter and the next parameter is a non-quoted string, it will treat all characters to the end of the command line as part of the string. This may or may not be what was intended. For example, the replace string command takes two string parameters, a string to search for and one to use as replacement. This command could be executed in the following ways.

(1) RS 'ABC' BCD or (2) RS 'ABC' 'BCD' or (3) RS ABC BCD

Examples 1 and 2 are equivalent and would execute as intended. However, in example 3, the editor would use ABC BCD as a single string and then prompt for the next string.

In some circumstances, the quoted string is different from the unquoted string. The editor uses all characters in an unquoted string as they appear. However, it gives special meaning to certain characters in a quoted string. The # symbol is used to signify that a two character hex digit follows. The editor converts such 3 character sequences into a single character. For example, '#41' is converted to the single character A. The = symbol is used to signify that a two character command mnemonic follows. The editor converts this 3 character sequence into an internal command code. For example, '=RS' is converted to the internal editor code for the replace string command. The ^ symbol is used to represent the CTRL key. When this symbol is encountered, the editor converts the next printable character into the corresponding non-printable control character. For example, '^Q' is converted to the single character, CTRL Q. If one of these special symbols is needed as a character in a quoted string, the symbol must appear twice. For example, '==' is equivalent to =. The sequence '^^' however represents CTRL ^. (use #5E) Representation of the quote character itself within a string is handled in the same manner. For example, the string '''' represents a single quote character.

In certain situations, such as defining a macro command (discussed in the following chapter), it may be necessary to use a quoted string within another quoted string. The editor accepts both single and double quotes as string delimiters. The case where you need a string within a string may be handled by using double quotes to delimit the outermost string, and single quotes to delimit the inner strings. (eg. "RS 'cba' 'abc'")

B. Editor State Commands

The commands listed in this section allow specific editor states to be set. Some of the commands require parameters while others do not. The commands which require no parameters act as switches that toggle a specific editor characteristic between two states. The commands requiring parameters are used to define values for the editor to use when that specific function is performed.

B.1 No Parameters

Mnemonic	Command Function
AI	Auto Indent - Turns auto indent on and off. The auto indent feature effects the positioning of the cursor when the <enter> key is pressed. If auto indent is off, the cursor is positioned at the left edge of the screen. If auto indent is on, the the cursor is positioned over the first non-blank character on the next line. If the line contains all blanks, the cursor is positioned under the first non-blank character on the line above. Default: off
CM	Command Mode - Toggles the editor between command mode and compose mode. Default: Compose mode
DT	Delete Tabs - Deletes all tab stops. Default: tab stops are set every 4 columns.
IC	Insert Character Mode - Used in overwrite mode to temporarily cause the editor to insert subsequent printable characters. Overwrite mode is reentered when a non-printable character is typed. Default: off
IM	Insert Mode - Toggles the editor between overwrite mode and insert mode. Default: Overwrite mode

Mnemonic	Command Function
LN	Lines - Toggles editor line numbering on and off. If line numbering is on, all lines of text in the buffer will be preceded by a line number. The line numbering is relative to the beginning of the file with the first line being number 1. The line numbers are not actually a part of the file. The editor simply maintains the line numbers internally and displays them for the text in the buffer. Default: off
TF	Tabify - Toggles blank compression on and off. This effects whether or not the editor outputs the tab character when the EX, XT, PR, or WR commands are executed. If compression is on, each sequence of 8 blanks in the editor buffer is converted to a single tab character before being written out. If compression is off, blanks are not converted to tab characters. With compression on, less space is required to store the file. However, some devices, such as printers, do not understand the tab character. Default: On

B.2 Parameters

Mnemonic	Command Function
RL	Roll - Sets the number of lines scrolled when the roll up (RU) or roll down (RD) commands are executed. Command Name: ROLL Number of Parameters: 1 Parameter Type: integer Example: <>ROLL 12 Default: (screen height) - (3)
TS	Tab Set - Defines the tab stops in either of two ways. The first way is to specify a single integer parameter. The editor will clear all current tab stops and set new stops beginning at column 1, with tab stops set every specified number of spaces. The second way is to specify the = symbol followed by a list of columns to be used as tab stops. Current tab stops are not deleted when this form of the command is used. The list of tab stops must be separated by commas. Command Name: TABS Number of Parameters: method 1 --> 1 method 2 --> n Parameter Type: integer Example method 1: <>TABS 10 Example method 2: <>TABS = 1,11,21,31,41,51,61,71 The above examples are equivalent. Default: <>TABS 4

C. Cursor Positioning Commands

The cursor positioning commands are commands which when executed result in the cursors position being changed. These commands cause the cursor to move by character, word, or line.

C.1 No Parameters

Mnemonic	Command Functon
BB	Bottom of Buffer - move cursor to the bottom of the text buffer.
BL	Beginning of Line - move cursor to the beginning of the line.
BT	Back Tab - move the cursor backward by one tab stop.
BW	BackWord - move the cursor backward by one word.
DN	Down - move the cursor down one line.
EL	End of Line - move the cursor to the end of the line.
FW	ForWord - move the cursor forward by one word.
GM	Go to Mark - move the cursor to the marked line. (used in conjuction with the mark (MK) block command)
HM	Home - move the cursor to the top left corner of the screen.
LF	Left - move the cursor left one character.
NL	Next Line - In overwrite mode, move the cursor to the beginning of the next line. In insert mode, insert carriage return at the current cursor position.
RD	Roll Down - scroll the text buffer window toward the end of the buffer.
RT	Right - move the cursor right one character.

Mnemonic	Command Function
RU	Roll Up - scroll the text buffer window toward the beginning of the buffer.
SW	Swap - swap the cursor line and the marked line. Moves the cursor to last line marked by the mark (MK) command.
TB	Tab - In overwrite mode, moves the cursor right to the next tab stop. In insert mode, moves the character under the cursor to the next tab stop by inserting blanks.
TP	Top - moves the cursor to the top of the text buffer.
UP	Up - moves the cursor up one line.

C.2 Parameters

HS	Horizontal Scroll - scrolls the screen horizontally so that the specified column is at the edge of the screen. (For terminals with less than 80 column wide screens only) Command Name: HSCROLL Number of Parameters: 1 Parameter Type: integer Example: <>HSCROLL 10
MI	Minus - move the cursor towards the top of the text buffer by a specified number of lines. Command Name: - Number of Parameters: 1 Parameter Type: integer Example: <>- 49
PL	Plus - move the cursor towards the bottom of the text buffer by a specified number of lines. Command Name: + Number of Parameters: 1 Parameter Type: integer Example: <>+ 35

Mnemonic	Command Function
PO	Position - move the cursor to the specified row and column on the screen. Top left corner of screen is row 0, column 0. Command Name: POSITION Number of Parameters: 2 Parameter Type: integer, integer Example: <>POSITION 5 10
SC	Set Column - move the cursor to the specified screen column. Command Name: COL Number of Parameters: 1 Parameter Type: integer Example: <>COL 13
SL	Show Line - positions the display so that the specified line is under the cursor. If the line is not currently in the text buffer, the editor automatically pages the original file into the buffer until the specified line has been loaded. If the specified line has already been written to the work file, then the editor will display the first line in the buffer at the top of the screen. Command Name: SHOWLINE Number of Parameters: 1 Parameter Type: integer Example: <>SHOWLINE 1000
SR	Set Row - moves the cursor to the specified screen row. Command Name: ROW Number of Parameters: 1 Parameter Type: integer Example: <>ROW 9

D. Character Commands

The character commands operate on a single character or a word.

Mnemonic	Command Function
DC	Delete Character - Deletes the character under the cursor.
DW	Delete Word - Deletes the word under the cursor. If the cursor is on a non-blank character, then all characters in both directions from the cursor to the first blank character are deleted. If the cursor is on a blank character, then all consecutive blank characters from the cursor to the right are deleted.
QU	Quote Character - After execution, the next character typed will be entered into the text at the current cursor position. This command may be used to insert non-printable control characters into the text buffer. caution: Some terminals treat certain non-printable characters as commands. If a non-printable character is recognized as a command by the terminal, the editor will not be able to display the text properly. Never insert a CNTL Z into the text. The editor uses CNTL Z for end of file.
RB	Rub Out - Deletes the character preceding the cursor.

E. Line Commands

The line commands operate on a either one or two lines at a time.

Mnemonic	Command Function
CL	Center Line - Centers the current cursor line on the screen.
DE	Delete to End of Line - delete all characters from the cursor to the end of the line.
DL	Delete Line - Delete the entire cursor line.
DU	Duplicate Line - Duplicates the line above the cursor onto the cursor line. Only the characters from the current cursor position to the end of the line are duplicated.
IL	Insert Line - Inserts a blank line at the current cursor position.
MG	Merge Line - Merges the line following the cursor to the end of the cursor line. Only the amount of text that will fit an 80 character line is merged.
SP	Split Line - Splits the current line at the cursor by inserting a blank line after the cursor line and placing the text from the cursor to the end of the line on the blank line.
UL	Undelete Line - Restores the last deleted line at the current cursor position. The editor maintains a line buffer which is initially filled with blanks. Each time a line is deleted (DL command), the deleted line is saved in this buffer. This command inserts the line from the buffer prior to the current cursor position.

F. String Commands

The string commands operate on specified character sequences. A character sequence cannot span a line boundary. The case of a character in the sequence is significant.

F.1 No Parameters

Mnemonic	Command Function
FN	Find Next - Find the next occurence of the "find string" in the text buffer. The "find string" is defined by the FS command. The search for the "find string" begins one character right of the cursor and continues till the string is found or the end of the text buffer is encountered.
RN	Replace Next - Find the next occurrence of the "find string" and replace it with the "replace string". The "find string" and "replace string" are defined by the RS command. The search for the "find string" begins one character right of the cursor and continues till the string is found or the end of the text buffer is encountered. If the "find string" is found, it is replaced by the "replace string".

F.2 Parameters

FS	Find String - Search from the current cursor position towards the end of the text buffer for the first occurrence of the specified string of characters. The single string parameter defines the "find string" buffer. Command Name: FIND Number of Parameters: 1 Parameter Type: string Example: <>FIND end <>FIND '#65#6E#64' The examples are equivalent.

Mnemonic	Command Function
QS	Quote String - Enter the specified string into the text buffer at the current cursor position. Command Name: QUOTE Number of Parameters: 1 Parameter Type: string Example: <>QUOTE abc <>QUOTE '#61#62#63' The examples are equivalent.
RG	Replace Global - Replace all occurrences of the "find string" with the "replace string". The search starts at the current cursor position and continues to the end of the text buffer. The two string parameters define the "find string" and "replace string" buffers. Command Name: REPGLOB Number of Parameters: 2 Parameter Type: string Example: <>REPGLOB 'recieve' 'receive'
RS	Replace String - Replace the first occurrence of the "find string" with the "replace string". The search starts at the current cursor position and continues until the "find string" is found or the end of the text buffer is encountered. The two string parameters define the "find string" and "replace string" buffers. Command Name: REPLACE Number of Parameters: 2 Parameter Type: string Example: <>REPLACE '132' '123'

G. Block Commands

The block commands operate on a block of text at a time. A block is a group of consecutive lines of text. All the block commands except for UR and LR operate on line boundaries. (ie. the column position of the cursor is not important). A block region must be marked (MK) before any block operations may be performed. All block operations apply to the text between the marked line and the cursor line (inclusive).
Some of the block commands use a temporary work file for buffer storage. The editor automatically creates this work file. The command explanations below refer to "block buffer". The temporary work file is used as the block buffer.

G.1 No Parameters

Mnemonic	Command Function
CB	Copy Block - Copies the text between the marked line and the cursor line to the block buffer.
DB	Delete Block - Deletes the text between the marked line and the cursor line.
IB	Insert Block - Inserts the text from the block buffer into the text buffer just prior to the cursor line.
LR	Lower Case - Converts all characters between the marked character and the character under the cursor to lower case.
MK	Mark - Places an invisible mark at the current cursor position. This command is used to define one block boundary. The other boundary is defined by the cursor position.
PR	Print - Print the text between the marked line and the cursor line. This command requires that a printer be connected to the computer.
UR	Upper Case - Converts all characters between the marked character and the character under the cursor to upper case.

G.2 Parameters

Mnemonic	Command Function
FI	Fill - Fills the marked block of text such that all text within the block fits inside the specified left and right margins. Command Name: FILL Number of Parameters: 2 Parameter Type: integer Example: FILL 10 70
JF	Justify - Justifies the marked block of text such that each line in the block aligns with the other lines at both specified left and right margins. Command Name: JUSTIFY Number of Parameters: 2 Parameter Type: integer Example: JUSTIFY 15 65
XT	Extract - Write the text within the marked block to the specified file. The file may be any legal file name, including drive specifier. Command Name: EXTRACT Number of Parameters: 1 Parameter Type: string Example: EXTRACT datafile

H. File Commands

The file commands are disk related commands. Most of the commands listed in this section result in text being written to or read from a disk file. All the file commands require parameters. The file parameters may be specified using any valid system file name, including drive specifier and user number if applicable. The two file commands APPEND (AP) and WRITE (WR) must be used when editing files larger than the text buffer.

For some commands, the editor uses a default file name if the file name prompt is answered simply by typing the <enter> key. If the editor is executed with a file specified on the command line, this file becomes the default file. Another way of defining a default file is to execute the editor without specifying a file on the command line and then using the APPEND (AP) command to load in a file. The file specified in the append becomes the default file. When the EXIT (EX) or SAVE (SV) commands are executed, the file prompts may be answered by simply typing <enter>. This will cause the default file to be used.

Mnemonic	Command Function
AP	Append - Appends a specified number of lines to the end of the text buffer. A second file parameter is optional. If no file is specified, then the editor uses the default file if it exists. Otherwise, the editor prompts for the file parameter. Multiple files may be appended using this command. Any appended file must be completely loaded (ie. EOF has been reached) before another file may be appended. When an exit is performed, the last file opened by an append will be completely written to the editor work file, even if not all the file was loaded into the text buffer. An exit without an explicit file specification will cause the editor to use the first appended file as the default file. Command Name: APPEND Number Parameters: 2 Parameter Type: integer, <string> Example: <>APPEND 100 <>APPEND 50 filex

Mnemonic	Command Function
DI	Directory - Displays the file directory of the specified drive. If no drive is specified, the default drive is used. The message - DONE - will appear at the bottom of the screen when all files have been listed. Typing any key will cause the editor to resume. Command Name: DIR Number of Parameters: 1 Parameter Type: <string> Example: <>DIR
EX	Exit - Causes the editor to perform an exit, returning control to the operating system. The current file being edited will be saved in the specified file or in the default file if no file is specified. The editor then prompts as to whether or not to create a backup. By default, pressing the <enter> key, a backup will be saved. If the file is a new file, no backup file is created. If the edited file already existed, then it is renamed unchanged as the backup. The backup file then reflects the file contents just prior to the last edit. The backup file is named by changing the extension of the original file to BAK. Command Name: EXIT Number of Parameters: 2 Parameter Type: string, Y/N Example: <>EXIT 'FILEA' Y
E/	Exit - Identical to EX above except that the exit returns back to the editor rather than the operating system. After the exit has finished, the text buffer is cleared and another file may be edited by using the APPEND command (AP). Command Name: EXIT/ (see EX)

Mnemonic	Command Function
HP	Help - (see general commands)
IF	Insert File - Insert a file or portion of a file into the text buffer just prior to the cursor line. The line number in the file where the insertion will start, followed by the number of lines to insert must be specified. The first line of a file is line number 1. If the number of lines specified is greater than the length of the file, then all lines from the starting line to the end of the file will be inserted. Command Name: INSFILE Number of Parameters: 3 Parameter Type: string, integer, integer Example: <>INSFILE 'SEGMENT' 1 45
QT	Quit - Causes the editor to exit to the operating system without saving the file. All editing is lost. This command provides a prompt to allow the user to abort the operation and reenter the editor. Command Name: QUIT Number of Parameters: 1 Parameter Type: Y/N Example: <> QUIT Y
Q/	Quit - Identical to QT above, except rather than exit to the operating system, the text buffer is cleared and you may continue editting. Command Name: QUIT/ Number of Parameters: 1 Parameter Type: Y/N Example: <>QUIT/ Y
SD	Swap Disk - This command tells the editor that you have swapped disks. It allows you to remove a disk and insert another during an edit session. Each time a disk is swapped, this command must be executed. If this command has been executed, the editor will prompt with DISK RESTORED? when an exit is attempted. If you type Y, then the exit is continued. Otherwise, the edit session is resumed. Command Name: SWAPD Number of Parameters: none Example: <>SWAPD

Mnemonic	Command Function
SF	Show File - This command allows a specified file to be displayed during an edit session. When the command is executed, the specified file is opened and a screenfull of lines is displayed, the top line on the screen displaying the first line in the file. Command Name: SHOWFILE Number of Parameters: 1 Parameter Type: string Example: <>SHOWFILE FILEY

Several commands are available to view the file when the SF command is executed. Three of these commands are standard editor commands which must be mapped to keys in order to execute them. Three others are special commands used only while in SHOWFILE.

Standard Editor Commands

Mnemonic	Key	
RU	CTRL R	- Scrolls one screenfull toward the beginning of the file.
RD	CTRL C	- Scrolls one screenfull toward the end of the file.
CM	CTRL Z	- Terminates SHOWFILE and resumes editing.

Special SHOWFILE Commands

<linenumber>	- By simply typing an integer line number, followed by the <enter> key, the specified line will be displayed on the top line of the screen.
+<lines>	- By typing the + symbol followed by an integer number, the line displayed at the top of the screen will be the current line number added with the specified integer.
-<lines>	- By typing the - symbol followed by an integer number, the line displayed at the top of the screen will be the current line number subtracted by the specified integer.

Mnemonic	Command Function
SV	Save File - This command saves the current edit session to the specified file and then resumes the edit. It is identical to the EXIT (EX) command except that rather than exit to the operating system, it resumes the current edit session. This command may be used to periodically save the text being edited. The second prompt asks whether or not to create a backup. Command Name: SAVE Number of Parameters: 2 Parameter Type: string, Y/N Example: <>SAVE 'AFILE' Y
WR	Write - This command writes a specified number of lines to the main editor work file. The specified number of lines are written starting with the first line in the text buffer. The lines are appended to the end of the main editor work file. All lines written are then deleted from the text buffer. This command may be used in conjunction with the append command (AP) to edit files larger than the text buffer. Command Name: WRITE Number of Parameters: 1 Parameter Type: integer Example: <>WRITE 100
XT	Extract - (see block commands)

I. General Commands

I.1 No parameters

Mnemonic	Command Function
CT	Clear Tab - Clears the tab stop, if one is set, at the current cursor position.
MM	Memory - Displays at the bottom of the screen, the amount of space remaining in the text buffer. The first number displayed is the number of characters which may be stored in the remaining space. The second number shows the amount of free space as a percentage of the total buffer space.
RF	Refresh - Causes the editor to redisplay the screen. This may be useful to determine whether or not non-printable characters are in the buffer. If the display behaves improperly, then the buffer contains a non-printable character which the terminal recognizes as a command.
ST	Set Tab - Sets a tab stop at the current cursor position.

I.2 Parameters

HP	Help - The help command displays the specified file. The only difference between the HELP (HP) command and the SHOWFILE (SF) command is that the help command appends the extension HLP to the specified file. The same commands may be used to view the file. The supplied files that have the extension HLP are editor help files which describe the editor commands. The parameter for the help command should specify the prefix for these file names. (ie. the HLP extension should not be specified.) The help file named KEY contains information showing how commands are mapped to keys. If the standard mapping is altered, a help file may be created which reflects the new mapping. Command Name: HELP Number of Parameters: 1 Parameter Type: string Example: <>HELP KEY

J. The Edit Command

The editor has a command that allows a second file to be edited without terminating the current edit session. When the editor is executed from the operating system it starts out at level 1. This is the only level ever used unless the EDIT (ED) command is executed. This command causes the editor to go to level 2. When level 2 is entered, the editor clears the text buffer and essentially begins a new edit session. The editor state remains unchanged except for the fact that a new file is being edited. The level 1 edit session is preserved with its current state at the time level 2 is entered. When level 2 is terminated by either the QUIT (QT) or EXIT (EX) commands, level 1 is reentered in its preserved state. The two levels are independent. (ie. the level 2 edit does not effect anything in the level 1 edit or vice versa).

Note: a maximum of 2 levels is allowed.

Mnemonic	Command Function
ED	Edit - Clears the text buffer and starts a new edit. The first parameter is for the file to edit. If a file is specified, the first 100 lines are loaded into the text buffer. If the <enter> key is typed for the file prompt, the level two edit will start with an empty buffer. The next four parameters define an edit window. They define the top row, bottom row, left column, and right column of the screen. The editor will use only this window to display text. The rest of the screen will be left undisturbed. This is a useful feature if you need to view a few lines of text from the level 1 edit while editing in level 2. If the specified left and right column parameters define a window too narrow to display all the text horizontally, the horizontal scroll command (HS) may be used to scroll the text in and out of the window. The window parameter prompts may be answered by typing the <enter> key. When the <enter> key is typed, the default value used is the screen boundary. Command Name: EDIT Number of Parameters: 5 Parameter Type: string, integer... Example: <>EDIT 'filez' 1 10 1 50

The editor has many commands which change specific characteristics of the editor. This chapter explains the two most powerful commands for changing the editor characteristics. One command allows you to change the way the keyboard is mapped to the editor commands. The other allows you to define more powerful editor commands. A single command may be constructed from the set of built in commands. This is called a macro command because it combines more than one of the built in commands to form a single more powerful command.

A. Translating Keyboard Characters to Commands

The editor is supplied such that selected commands are internally mapped to the keyboard. It is possible to change this mapping partially or completely to suit personal preference. The TRANS (TR) command will allow you to map any keyboard generated character sequence to any editor mnemonic. The character sequence consists of one or more ascii characters, printable or non-printable. It is best to start the sequence with a non-printable character (eg. the ESC character or a control character). If a printable character starts the sequence, you will no longer be able to enter that character as text. The editor treats all character sequences defined by the TRANS command as a single entity. As each character is received from the keyboard, it is checked to see if it begins a sequence mapped to an editor command. If not, then the character is entered as text if it is printable or discarded if it is non-printable. If the character does begin a defined sequence, the appropriate command is executed for the case of a single character sequence. For multiple character sequences, subsequent characters are used to determine the appropriate command. (ie. they are not entered as text) Only after the sequence traces to a specific command or an invalid sequence does the editor revert back to entering printable characters as text.

The TRANS command takes two string parameters. The first parameter is the character sequence. The second parameter is the editor command to which the sequence is mapped.

There are three special symbols which may be used in quoted strings. (#, ^, and =). The editor gives special meaning to these symbols. If the symbol itself is to be a character in the string, it must appear twice. For example, '##' would be the single character #.

Any ascii character may be represented in a quoted string. The non-printable characters may be represented by a three character sequence beginning with the # symbol. The two characters following the symbol must be a valid hexadecimal value. The ascii chart shows the hexadecimal value for each character in the ascii character set.

The ^ symbol is used to represent the control key, CTRL. For example, CTRL Q may be represented in a string as '^Q'. This is equivalent to the string '#11' since CTRL Q has the ascii value of hexadecimal 11. As you can see, '^Q' is a little more readable than '#11'. The character sequence CTRL Q, f, Z is represented as '^QfZ'. The sequence CTRL W, CTRL X is represented as '^W^X'. The escape character ESC may be represented as '^['.

The = symbol is used to signify that a two character editor command mnemonic follows. For example, the string '=UP' represents the cursor up command (UP). The string '=FS' represents the find string command (FS).

Mnemonic	Command Function
TR	Translate - Translates the specified character sequence to the specified editor command. The character sequence is the first parameter. It may consist of any number of valid ascii characters. The second parameter must be a built in editor command mnemonic. Command Name: TRANS Number of Parameters: 2 Parameter Type: string, string Example: <>TRANS '^R' '=RU' <>TRANS '^QR' '=TP' <>TRANS '^Q^R' '=TP'

While in command mode, rather than type ^ followed by a character, the actual control character may be typed. The editor will echo the two character representation. For example, if CTRL R is typed, the editor will echo ^R to the command line.

B. Defining Macro Commands

It is possible to map a keyboard generated character sequence to more than one editor command. The DEFINE (DM) command is similar to the TRANS command in that it maps a keyboard generated character sequence to an editor command. However, the DM command will allow you to specify more than just a single editor mnemonic.

The second parameter of the DM command is a string which may contain command mnemonics (including parameters if required), command mapped character sequences, or a sequence of printable characters. The editor uses this string as input just as it normally does with input from the keyboard. Essentially, anything that may done manually from the keyboard may be defined in this string and executed automatically.

A simple illustration of the use of the define command is to map a key to a word which is often typed. The following example maps the word Program to the key sequence ESC P (ie. the escape key followed by upper case P). Then typing ESC P is equivalent to typing the word Program.

```
<>DEFINE '^[P' 'Program'
```

Commands may be specified in several ways. One way is to specify the command mnemonic preceded by the = symbol. If the command requires parameters, the parameters should immediately follow the mnemonic. Each parameter must be followed by the NL command as a parameter terminator. The NL command is mapped to the <enter> key which is equivalent to CTRL M. Therefore, either ^M or =NL may be used as a parameter terminator. The following macro will move the cursor forward by sentence. The example maps the macro to ESC S. When executed, the find string command positions the cursor over the next period in the text. Then the right cursor command is used to handle cases where the period is the last character on a line. The forward word command then causes the cursor to be positioned at the beginning of the next sentence.

```
<>DEFINE '^[S' '=FS.^M=RT=FW'
```

The above command is executed from the editor compose mode. First the =FS command is executed which causes the editor to prompt for the <FIND>STRING: parameter. The editor then uses the sequence of characters up to the terminator as the find string parameter. At this point, the find string command is executed. When finished, the next input received is the cursor right and forward word commands. After the forward word command is executed, control returns to the keyboard.

The following example illustrates another way of defining the macro to move forward by sentence. First the editor is switched to command mode. The find string command follows, terminated by the next line command. Note that once in command mode, the command syntax is identical to that when executing manually from command mode. When the find string command finishes execution, the editor returns to compose mode. Then the cursor right and forward word commands are executed. Here, the RT and FW commands are specified as the control characters to which they are mapped rather than by the mnemonics.

```
<>DEFINE '^[S' "=CMFIND '.'=NL^D^F"
```

Any command mapped character sequence may be used in defining a macro. This means that one macro may reference another defined macro. Since strings may not cross line boundaries, this provides a way of building macros longer than one line. The following example illustrates a macro definition which refers to another defined macro.

Suppose you wish to define a macro to capitalize the first character of the current word under the cursor. First, define a macro which positions the cursor to the beginning of the current word. The back word command will do this. However, if the cursor is already on the first character of a word, the back word command will cause the cursor to be positioned at the beginning of the previous word. To prevent this from occurring, position the cursor right one character before performing the back word command.

```
<>DEFINE '^[$' '=RT=BW'
```

Now you can define a macro which capitalizes the character under the cursor. The UR block command may be used to do this. First, the mark command is used to place an invisible mark at the cursor position. Since the cursor is at the same position as the mark, only the character under the cursor will be capitalized.

```
<>DEFINE '^[@' '=MK=UR'
```

A macro can now be defined which uses the above defined macros. CTRL W can then be used to capitalize the current word under the cursor.

```
<>DEFINE '^W' '^[$^[@'
```

This particular macro is short enough to have been defined as a single macro. However, it does illustrate how one macro may use another defined macro.

The maximum number of macros that may be defined at any one time is 64. If you wish to change the macro defined to a particular character sequence, you may simply use the DM command to define a new macro to that particular sequence. However, when this is done, the memory required to store the previously mapped macro will not be recovered. To recover the memory used by the old macro, the undefine macro command (UM) may be used. This command will cause the macro defined by the specified character sequence to be deleted.

Mnemonic	Command Function
DM	Define Macro - Maps keyboard generated character sequences to editor commands, other command mapped sequences, printable text, or any combination of the above. Command Name: DEFINE Number of Parameters: 2 Parameter Type: string, string Example: <>DEFINE '^W' 'this is a macro'
UM	Undefine Macro - Recovers the space used by a macro definition. The macro mapped to the specified character sequence is deleted. Command Name: UNDEFINE Number of Parameters: 1 Parameter Type: string Example: <>UNDEFINE '^W'

The editor must use a setup file at a minimum to determine the terminal characteristics. In addition, the setup file may be used to customize the editor by setting specific editor states, mapping commands to keys, and defining new commands. Several setup files may be created to allow the editor to be configured differently, depending on the type of editing being performed. Then simply specify the appropriate setup file when the editor is executed. You might use one setup file for programming and another one for word processing. If you use more than one programming language, you might have a separate setup file for each language.

There are some normal editor commands which are allowed in setup files and some special commands which can be used only in setup files. Among the special commands are four terminal defining commands (HEIGHT, WIDTH, TERMINAL, and CURSOR). These are the commands which the SETEDIT utitity outputs to a setup file to define terminal characteristics. When creating a setup file, you may exclude the terminal characteristics. After creating a text format setup file using the editor, the SETEDIT utility may be used to read it, merge it with the terminal information, and then write the combined information to a binary and/or text format file. The terminal information may be merged by selecting the proper terminal from the menu or by reading a previously created setup file containing the terminal information.

Remember, the editor requires binary setup files. However, the text format setup file is useful if you wish to make modifications.

The commands in setup files are limited to one command per line. The semicolon ; may be used as a comment specifier. If a semicolon is encountered in a setup file, the remaining text on that line is treated as a comment. This of course does not apply to semicolons which appear inside quoted strings. The commands may be placed in any order within the setup file. The only requirement is that a command mapped character sequence must be defined before it is referenced by the START or DEFINE commands.

A. Normal Commands

This section lists the normal editor commands which may be used in a setup file. In setup files, these commands must be specified using the command name. The mnemonic is not allowed. Otherwise, the form is the same as described in chapters 4 and 5.

A.1 TABS

The TABS command may be used to change the default tab setting. The default is TABS 4. Both forms of the TABS command may be used. See the TS command in the editor state commands.

A.2 ROLL

The ROLL command may be used to change the default setting for the number of lines scrolled by the roll up and roll down commands. The default roll size is three less than the screen height. See the RL command in the editor state commands.

A.3 AUTOINDENT

The AUTOINDENT command may be used to turn on auto indent from the setup file. The default is off.

A.4 TRANS

The TRANS command may be used to change the default mapping of commands to keys. A complete remapping may be performed or the default mapping may be slightly modified. See the TR command in chapter 5.

A.5 DEFINE

The DEFINE command may be used to define macro editor commands formed from the built in commands. If a macro command references a character sequence defined by the TRANS command or by another DEFINE command, the referenced sequence must have been defined on a previous line. There is a limit of 64 macro definitions. See the DM command in chapter 5.

B. Special Commands

The special commands are commands which may only be used in a setup file. The last four commands in this section define terminal characteristics. These commands are automatically created by the SETEDIT utility and therefore may be included in the setup file through the use of this utility.

B.1 The INIT Command

The INIT command may be used to send a string of characters to the terminal when the setup file is loaded. The command takes one parameter which is a quoted string. The string may contain either printable or non-printable characters. Printable characters may be sent to identify the setup file being used for the current edit session. Non- printable characters which the terminal intercepts as commands may also be used to set some desired terminal characteristic.

Example: INIT 'Pascal Setup File'

B.2 The EXIT Command

The EXIT command is identical to the INIT command except the string is not sent to the terminal until the editor is exited. There can only be one EXIT command in a setup file.

Example: EXIT 'Edit Finished'

B.3 The START Command

The START command specifies a string of commands that the editor executes immediately after the file to be edited has been loaded. This command may be used to execute other editor commands which are not allowed in the setup file. The following example causes the editor to come up in insert mode with compression turned off. There can only be one START command in a setup file.

Example: START '=IM=TF'

B.4 The CMD Command

The CMD command provides a way of giving names to the built in editor commands. The commands which do not require parameters have only a two character mnemonic. The CMD command may be used to define a longer name to also be associated with a particular command. It requires two parameters. The first is the name identifier. The second is a string containing an editor mnemonic. This may make it easier to remember than the two character mnemonic. The following example assigns the name MARK to the mnemonic MK. This will allow the mark command to be executed by either the full name or the two character mnemonic when in command mode.

Example: CMD MARK '=MK'

The following four commands describe the terminal characteristics. The SETEDIT utility may be used to create these commands in a setup file.

B.5 The HEIGHT Command

The HEIGHT command takes a single integer parameter which defines the number of lines on the screen.

Example: HEIGHT 24

B.6 The WIDTH Command

The WIDTH command takes a single integer parameter which defines the character width of the screen.

Example: WIDTH 80

B.7 The TERMINAL Command

The TERMINAL command defines the features of the terminal. It takes two parameters. The first is an identifier which identifies a particular function of the terminal. The second parameter is a string containing the character sequence required by the terminal to perform that specific function.
The editor makes use of most of the smart functions included in many of the latest terminals.

The following features are supported.

CLEAR - clear screen
CLREOS - clear to end of screen
CLREOL - clear to end of line
INSLINE - insert line
DELLINE - delete line
DELCHAR - delete character
INSMODE - enter insert mode
NOINS - exit insert mode
RSCROLL - scroll the screen 1 line down
(reverse linefeed with cursor at top of screen)
SCROLL - scroll the screen 1 line up
(linefeed with cursor at bottom of screen)
INSONE - insert one character

Other parameters of the terminal command specify how cursor addressing is performed.

CURSOR - specifies the character sequence which precedes the row and column. This parameter is used if the terminal does not require character sequences between and following the row and column address.

CURSOR1 - specifies the character sequence which precedes the row and column. This parameter is used if the terminal requires character sequences between and following the row and column address.

CURSOR2 - specifies the character sequence which must appear between the row and column.

CURSOR3 - specifies the character sequence which follows the row/column address.

COFFSET - specifies the offset for addressing the first row or column on the screen.

Example: TERMINAL CLEAR '^[Y'

B.8 The CURSOR Command

The CURSOR command takes one parameter which describes the algorithm used to address the cursor. The following is a list of the possible cursor addressing methods. The SETEDIT utility will generate one of these adressing methods.

ROWCOL, ANSII, COLROW, BINARY, ASCII, SPECIAL

Example: CURSOR ROWCOL

C. Sample Setup File

```
;*************************************************************
;*            TRS-80 MODEL 4 SAMPLE SETUP FILE               *
;*  This file is supplied on disk (SAMPLE/EDT) in binary     *
;*  format and may be used in place of SETUP/EDT if desired.*
;*  Among other things, it maps the arrow keys to            *
;*  appropriate cursor movement commands.  To use this file,*
;*  first rename SETUP/EDT and then rename SAMPLE/EDT to     *
;*  SETUP/EDT.  The SETEDIT utility may be used to create    *
;*  a text format file if you wish to modify this setup      *
;*  file.              -----------------                     *
;*     This setup file maps some commands to the clear       *
;*     and break keys.  In the explanations below, the       *
;*     clear key is represented as <CLR> and the break       *
;*     key as <BRK>.  When executing commands mapped with    *
;*     <CLR>, the clear key should be held down.  When       *
;*     executing commands mapped with <BRK>, the break       *
;*     key should be pressed and released.                   *
;*     The keys which are mapped to commands are commented   *
;*     in the form:   ;key                --> command       *
;*  Note:                                                    *
;*     The editor's internal mapping of keys to commands     *
;*     remains valid for all keys which are not explicitly   *
;*     remapped by this setup file.                          *
;*************************************************************
;------------------terminal definition ----------------------
;          (This section was created by SETEDIT)
;
TERMINAL CLEAR    '^\^_'          ;clear screen
TERMINAL CLREOS   '^_'            ;clear to end of screen
TERMINAL CLREOL   '^^'            ;clear to end of line
TERMINAL SCROLL   '^J'            ;scroll 1 line up
CURSOR   SPECIAL                  ;special cursor addressing
HEIGHT     24                     ;number of lines/screen
WIDTH      80                     ;number of characters/line
;------------- end of terminal definition ------------------
;
;               (Customization Section)
;
;     The following two commands send strings of characters
;     to the terminal.  ^N in the INIT string insures that
;     the Model 4 cursor is turned on.
;
INIT '^NReading Setup File'       ;send message at start
EXIT 'Edit Session Finished'      ;send message at end
;
;KEY TRANSLATIONS
;----------------
;     The following key tranlations redefine how editor
;     commands are mapped to the Model 4 keyboard.
;     Appendix B of the Model 4 Disk System Owners's
;     Manual has a keyboard diagram which shows the
;     characters generated by each key.
;               --------------------------
;     The following key translations map the arrow keys
;     to cursor movement commands.  The left arrow key
;     generates ^H which is internally mapped to =LF.
```

```
TRANS '^I'   '=RT'      ;right arrow         --> cursor right
TRANS '^J'   '=DN'      ;down arrow          --> cursor down
TRANS '^K'   '=UP'      ;up arrow            --> cursor up
TRANS '#8A'  '=RD'      ;<CLR> down arrow    --> roll down
TRANS '#8B'  '=RU'      ;<CLR> up arrow      --> roll up
TRANS '#88'  '=BT'      ;<CLR> left arrow    --> back tab
TRANS '#89'  '=TB'      ;<CLR> right arrow   --> tab
;
;  The following key translations map various commands
;  mnemonically using the clear key as a control key.
;  Use <CLR> N for insert line.
;
TRANS '#C6'  '=FN'      ;<CLR>  F            --> find next
TRANS '#D2'  '=RN'      ;<CLR>  R            --> replace next
TRANS '#C3'  '=DC'      ;<CLR>  C            --> delete character
TRANS '#D7'  '=DW'      ;<CLR>  W            --> delete word
TRANS '#CC'  '=DL'      ;<CLR>  L            --> delete line
TRANS '#D5'  '=UL'      ;<CLR>  U            --> undelete line
TRANS '#C4'  '=DU'      ;<CLR>  D            --> duplicate line
TRANS '#C9'  '=IC'      ;<CLR>  I            --> insert character
;
;  The following key translations map commands to the
;  3 function keys F1, F2, and F3. The shifted function
;  keys are represented as <SFn>
;
TRANS '#81'  '=CM'      ;<F1>                --> command mode
TRANS '#82'  '=BW'      ;<F2>                --> backward by word
TRANS '#83'  '=FW'      ;<F3>                --> forward by word
TRANS '#91'  '=IM'      ;<SF1>               --> insert mode
TRANS '#92'  '=SP'      ;<SF2>               --> split line
TRANS '#93'  '=MG'      ;<SF3>               --> merge line
;
;  The following key translations map commands to the
;  keys using the break key <BRK> as a prefix.
;
TRANS '#80#0A'  '=BB'   ;<BRK> down arrow    --> bottom of buffer
TRANS '#80#0B'  '=TP'   ;<BRK> up arrow      --> top of buffer
TRANS '#80#08'  '=BL'   ;<BRK> left arrow    --> beginning of line
TRANS '#80#09'  '=EL'   ;<BRK> right arrow   --> end of line
TRANS '#80L'    '=DE'   ;<BRK> L             --> delete to end of line
TRANS '#80l'    '=DE'   ;<BRK> l             --> delete to end of line
;
;
;EDITOR STATE CONFIGURATION
;--------------------------
;  The following commands set default states for the
;  editor.
;
START  '=TF'                ;tab compression off
TABS   3                    ;set tabs every 3 spaces
AUTOINDENT                  ;turn on auto-indent
ROLL  23                    ;set scrolling to screen height - 1
;
;DEFINE LONG NAMES FOR THESE COMMANDS
;------------------------------------
;  The following commands define long names which may
;  be used while in command mode to execute these commands.
;
CMD MARK    '=MK'           ;mark   is equivalent to mk
CMD INDENT  '=AI'           ;indent is equivalent to ai
```

```
;
;DEFINE MACROS
;-------------
;  The following commands define macro's which map
;  Pascal keywords to the numeric keys using clear as a
;  control key.
;
DEFINE '#B1' 'PROGRAM '                    ;<CLR> 1 --> PROGRAM
DEFINE '#B2' 'CONST '                      ;<CLR> 2 --> CONST
DEFINE '#B3' 'TYPE '                       ;<CLR> 3 --> TYPE
DEFINE '#B4' 'VAR '                        ;<CLR> 4 --> VAR
DEFINE '#B5' 'PROCEDURE '                  ;<CLR> 5 --> PROCEDURE
DEFINE '#B6' 'FUNCTION '                   ;<CLR> 6 --> FUNCTION
DEFINE '#B7' 'BEGIN '                      ;<CLR> 7 --> BEGIN
DEFINE '#B8' 'END'                         ;<CLR> 8 --> END
;
;  The following commands define macros's which use
;  the macro's defined above to create Pascal program shells.
;  The next line command (=NL) is internally mapped to ^M
;  which is generated by the <enter> key.  ^M is used in
;  place of =NL in the definitions below.
;
DEFINE '#80D' '#B2^M#B3^M#B4'              ;<BRK> D --> declarations
DEFINE '#80B' '#B7^M#B8'                   ;<BRK> B --> body
DEFINE '#80F' '#B6^M#80#44^M#80#42;'       ;<BRK> F --> function
DEFINE '#80P' '#B5^M#80#44^M#80#42;'       ;<BRK> P --> procedure
;
;  The following macro definition defines
;  <BRK> S to put a complete program shell on the screen
;
DEFINE '#80S' '=IM=BL^M^K#B1^M#80D^M^M#80P^M^M#80F^M^M#80B.=HM=FW =[illegible]'
;
;  The following definitions make the above macros work
;  with lower case.
;
DEFINE '#80d' '#80D'                       ;<BRK> d --> <BRK> D
DEFINE '#80b' '#80B'                       ;<BRK> b --> <BRK> B
DEFINE '#80f' '#80F'                       ;<BRK> f --> <BRK> F
DEFINE '#80p' '#80P'                       ;<BRK> p --> <BRK> P
DEFINE '#80s' '#80S'                       ;<BRK> s --> <BRK> S
;
;  The following macros define keys which terminate an edit
;  session.  <BRK> E is defined to exit and save the file
;  being edited but not save a backup file.  <BRK> Q is
;  defined to quit the edit without saving the file.
;
DEFINE '#80E' '=EX=NLN=NL'                 ;<BRK> E --> exit
DEFINE '#80Q' '=QTY=NL'                    ;<BRK> Q --> quit
DEFINE '#80e' '#80E'                       ;<BRK> e --> <BRK> E
DEFINE '#80q' '#80Q'                       ;<BRK> q --> <BRK> Q
;
; end of setup file
```

A. Sample Custom Terminal Setup

This is a sample execution of the SETEDIT utility using the CUSTOM terminal selection. The terminal used in the sample is the TELEVIDEO 925/950. Note that for steps 26 and 30, ^@ was used. This is the null character. Null characters are ignored by most terminals. They may therefore be used as fill characters in order to control timing. If a particular terminal function responds too slowly, null characters may be used to allow the terminal time to complete the function.

```
 1) Type SETEDIT
 2) Please make a selection: T <enter>
 3) Please select a terminal or 0 to exit: 31 <enter>
 4) Do you want to continue? Y <enter>
 5) B=binary, A=ascii: B <enter>
 6) Which is first, row or column (R,C): R <enter>
 7) enter a decimal number (space=32): 32 <enter>
 8) What characters come before the row number: ^[= <enter>
 9) What characters come between the row and column: <enter>
10) What characters come after the column number: <enter>
11) Does your terminal have clear screen        ? Y <enter>
12) Sequence to perform it: ^[+ <enter>
13) Does your terminal have clear to end screen ? Y <enter>
14) Sequence to perform it: ^[Y <enter>
15) Does your terminal have clear to end of line? Y <enter>
16) Sequence to perform it: ^[T <enter>
17) Does your terminal have insert line         ? Y <enter>
18) Sequence to perform it: ^[E <enter>
19) Does your terminal have delete line         ? Y <enter>
20) Sequence to perform it: ^[R <enter>
21) Does your terminal have delete character    ? Y <enter>
22) Sequence to perform it: ^[W <enter>
23) Does your terminal have enter insert mode   ? N <enter>
24) Does your terminal have exit insert mode    ? N <enter>
25) Does your terminal have scroll 1 line down  ? Y <enter>
26) Sequence to perform it: ^[j^@^@^@^@ <enter>
27) Does your terminal have insert 1 character  ? Y <enter>
28) Sequence to perform it: ^[Q <enter>
29) Does your terminal have scroll 1 line up    ? Y <enter>
30) Sequence to perform it: ^[J^@ <enter>
31) Please make a selection: W <enter>
32) Enter name for binary setup file: SETUP.EDT
33) Please make a selection: E <enter>
```

SYSTEM IMPLEMENTATION MANUAL

Table of Contents

Introduction..... 1

System Description..... 3
- PASCAL..... 3
- OPTIMIZE..... 3
- CODEGEN..... 3
- RUNP..... 3
- LINKLOAD..... 4
- PASCALB..... 4

Using TRS-80 Pascal..... 5
- Compiling a program..... 5
- The PASCAL command..... 6
- The RUNP command..... 7
- The compiler listing..... 8

Using the LINKLOAD program..... 9
- The Load command..... 10
- The Symbols command..... 10
- The Run command..... 11
- The Build command..... 11
- The Init command..... 12
- The Trsdos command..... 12
- Error messages..... 12

TRSDOS File and Device Names..... 13
Estimating Stack Size..... 14
Pascal Memory Usage..... 15
Compiler Memory Constraints..... 15
Real Numbers..... 16

Procedure and Function Library..... 17
- TRSLIB routines..... 18
  - System interface routines..... 18
  - Input and output routines..... 22
  - File routines..... 23
  - Screen routines..... 27
  - Extended memory routine..... 29
- RANDOM access file routines..... 31
  - Openrand..... 31
  - Readrand..... 31
  - Writerand..... 32
  - Closerand..... 32
- STRING routines..... 35

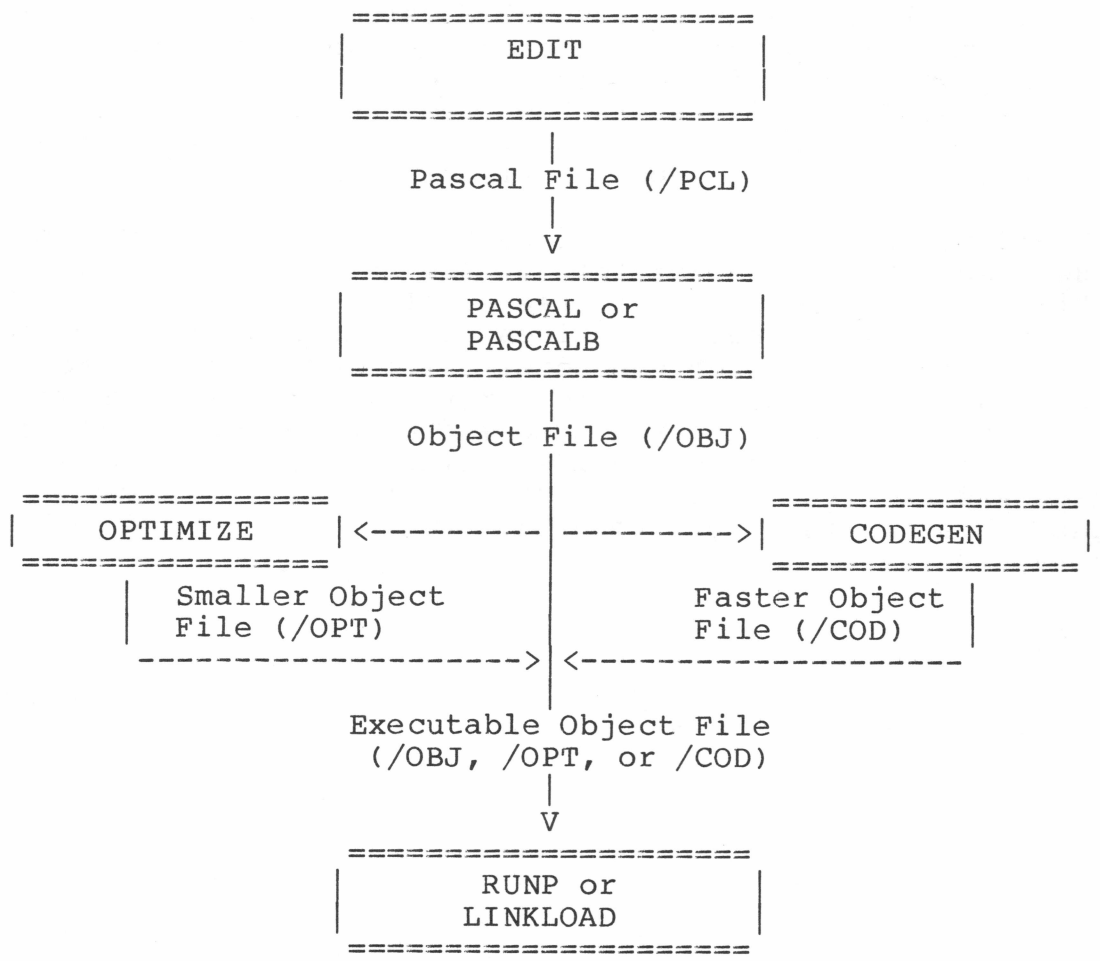
Linking Assembly Language to Pascal..... 37
Miscellaneous..... 39
- Generating EOF from the keyboard..... 39
- Patches..... 39

INTRODUCTION

This manual describes the specific characteristics of TRS-80 Pascal as implemented on the TRSDOS Version 6 operating system. In every language system implementation, there are certain language features that vary from one computer to the next. One of the advantages of TRS-80 Pascal is that these variations are minor. Machine dependent characteristics include such items as how to invoke the compiler and support utilities.

TRS-80 Pascal for the Model 4 is almost totally compatible with TRS-80 Pascal for the Model I/III. The only difference is the addition of a couple of procedures supplied in TRSLIB/OBJ. If use of these machine dependent routines is avoided, the object code from programs compiled on the Model I/III computers may be relinked with the proper runtime for the Model 4 and executed. The same applies for programs compiled on other computers using Alcor Pascal. They may be relinked with the TRS-80 runtime package and executed on the TRS-80. Portability between computer systems is a very important feature of this Pascal system.

System Overview Diagram



PASCAL

The Pascal compiler is simply a program that is written in Pascal and that executes on the host computer. It's purpose is to translate other Pascal source programs into an intermediate language called p-code. The p-code is a low level language designed specifically as a target language for the Pascal compiler and resembles the assembly language for a stack oriented computer. Once a program has been compiled, the object p-code program is stored as an object file (/OBJ). The /OBJ file may be executed directly or may be run through the advanced development package (ADP).

Advanced Development Package

OPTIMIZE (optional)

After the source program has been translated into object code, it may be processed by the optimizer. The purpose of the optimizer is to remove statement redundancy in the translated object code. This will effectively reduce the final size of the program by approximately 10-30 percent. The optimizer should be used where program size is important. The optimized p-code is an exceptionally compact representation of the Pascal program. This is evidenced by the fact that the Pascal compiler itself (an 8500 line Pascal program), can be run on a 48k machine without resorting to overlays.

CODEGEN (optional)

If program execution speed is important, the native code generator program may be used to process the object program. Codegen will generate native Z-80 code which may be directly executed by the processor. Execution speed is usually increased by a factor of 3 - 5 times. One of the drawbacks of code generation is that the resultant program will grow in object code size by a factor of 2 - 3 over the p-code version. For large Pascal programs, (such as the compiler itself) the resultant program image may not fit into available memory. For small programs this may not be a factor. To combine the best of both worlds, the codegen program will allow selective code generation of specific modules in a program. This allows the critical paths of a program to be translated into native Z-80 instructions, while at the same time reducing the overall program size by utilizing p-code for the rest of the program. If program size is not a factor, full code-generation may be performed.

RUNP

After the Pascal source program has been compiled, and/or processed by the advanced development package, it may be executed by the RUNP program. This utility will directly execute the compiled object code.

LINKLOAD

After the compiler has translated the source code into p-code, the p-code file may be loaded into memory and executed. The program that performs this is the LINKLOAD utility. Its purpose is to load any number of object modules into memory. This allows separate compilation of procedures and functions. To perform separate compilation of a procedure or function, the compiler NULLBODY option must be used. For more information, see the Reference Manual. The linking loader includes an interpreter in the final load module that executes the p-code instructions when the program is run. The linking loader also has the capability of storing the memory image of the program as an executable command file. Once an image has been saved, the program can be executed simply by typing the file name at the TRSDOS command level.

PASCALB
(Overlayed compiler)

The size of a Pascal program that may be compiled is dependent on the number of symbols used in the source program and not necessarily the number of lines in the program. The non-overlayed compiler (PASCAL) should be able to compile a typical 4000 line program with all of its associated symbols. A further improvement can sometimes be made by separately compiling procedures or functions and minimizing the use of global variables. If the program is too large for the non-overlayed compiler, the overlayed compiler may be used. The overlayed compiler has been segmented such that parts of it reside on the disk during execution, and are read into memory only as needed. The overlayed compiler will execute more slowly than the non-overlayed version, but generates identical object code. The overlayed compiler has enough space to compile a typical 10000 line Pascal program with all of its associated symbols.

This section describes the procedures for compiling and executing Pascal programs on the TRSDOS operating system. TRS-80 Pascal is designed to make this task as easy as possible. The first step is to analyze the problem to be solved and to write a Pascal program that solves it. There are many fine textbooks available that describe programming techniques. Pascal is a very powerful expression language for solving programming problems. If you are not familiar with the Pascal language, refer to the Tutorial Manual for information on the language. For those familiar with Pascal, the Reference Manual contains compact and detailed information on the features of TRS-80 Pascal.

Once the program has been designed, the next step is to enter the program into the computer. This is normally accomplished with the aid of a text editor. A screen oriented text editor is supplied with the compiler. For details on how to use this editor refer to the Editor Manual. Any other editor that can produce an ordinary Ascii text file may also be used.

COMPILING THE PROGRAM

When the program has been entered into the computer and placed in a disk file, the next step is to compile it. The Pascal compiler translates the source program into a form that the computer can execute. For example, suppose that you have developed a program to prepare your income tax return. This program may be stored in a file called: TAXES/PCL. The simplest method to compile and execute this program is to type the two commands:

```
                PASCAL TAXES <enter>     to compile the program
and             RUNP TAXES <enter>       to execute the program
```

Note: The PASCAL command appends the extension /PCL to the file name. The RUNP command appends the extension /OBJ to the file name if no extension is specified.

Let's examine the process in more detail. The first line causes the operating system to load and execute the Pascal compiler. The compiler then translates the Pascal source code contained in the file: TAXES/PCL into code that can be run on the computer. This code is stored in a file called: TAXES/OBJ . A listing will be sent to the screen. The listing shows the source program and will contain error messages for any errors detected. The listing will be described in more detail in a later section. If errors are detected, code numbers and error messages will be contained in the listing. The errors in the source program must be corrected before the program can be executed.

Once the program has been compiled without errors, it may be executed with the RUNP command. RUNP TAXES causes the object code stored in the file: TAXES/OBJ to be loaded into memory and executed.

The first thing that a Pascal program normally does is to open the logical files "INPUT" and "OUTPUT". When this happens, the prompts:

```
INPUT  =
OUTPUT =
```

will appear on the screen. At this time you may enter the file or device to be used when the program reads from input or writes to output. If you simply press the enter key, then input and output will be directed to the screen. When any file is opened by a Pascal program (by calls to RESET or REWRITE), a prompt will appear on the screen. To the left of the equal sign will be the Pascal name of the file being opened. You should type the name of the disk file or device to be associated with that file. Note - The INPUT, OUTPUT prompts may be eliminated by the use of the (*$NO INOUT*) option. See compiler options in the Reference Manual.

The runtime mapping of Pascal files to physical files and devices allows a program to redirect its input and output without any changes to the source program and without recompiling the program. For example, you could test the taxes program with the output going to the screen. When you are satisfied with the results, the output can be directed to a file or line printer instead.

The file names that you type to direct Pascal input and output are in the same format as normal TRSDOS file names. The disk drive specification is optional. Device names may also be substituted for filenames. Devices include :C (CRT), :L (line printer), and :D is a dummy device. If :D is used, no output will occur. This may be useful if you wish to discard some of the output of a program.

THE PASCAL COMMAND

The PASCAL command causes the Pascal compiler to be loaded and executed. This command has two forms. The simplest form is:

(angle brackets required when stack is specified)

```
PASCAL <stack>  filename
```

where filename is the name of a file containing a Pascal program. The <stack> is an optional parameter that sets an upper limit on memory space that the compiler may use for stack manipulations. The default stack size used by the compiler is 4K. This should be suitable for most applications. The compiler requires a minimum of 3.9K of stack to execute.

The compiler itself is a Pascal program and follows the same conventions for stack and heap usage as other Pascal programs (See pages 15,16). In the short form, the extension for the source file is assumed to be /PCL and the object code is sent to filename/OBJ. Any extension typed in the command line will be ignored. A disk drive name may also be specified. For example,

PASCAL TAXES:1 <enter>

will cause the program TAXES/PCL to be compiled and the object to be stored on disk drive 1. In this case the same disk drive will be used for both source and object. If the disk drive is omitted, the compiler will search for the file starting with drive 0. In the short form, the listing will always be displayed on the screen.

The long form of the Pascal command uses simply: PASCAL to invoke the compiler. In this case, the file names for the source, listing and object will be prompted for on the screen. You should type the names of the actual files to be used. Normal TRSDOS syntax applies. In this case the file names are used as specified. The source and object can be on different disk drives and the listing can be placed in a file, sent to the screen or sent to the line printer. For example, the following sequence will cause the file: TAXES/TMP to be compiled with the object code stored in TAXES/OBJ on disk drive 2. and the listing will be sent to the line printer.

```
PASCAL  <stack>
SOURCE  = TAXES/TMP
LISTING = :L
OBJECT  = TAXES/OBJ:2
```

THE RUNP COMMAND

The runp command is used to load and execute a previously compiled Pascal program. The object code will be loaded and the program executed. The runp command contains the object code for the TRS-80 support routines (such as PEEK, string routines, etc). Any of these routines can be called. If any other external procedures are required, the linking loader must be used to link these external procedures to the program. The runp command is invoked as:

RUNP TAXES <enter>

Pascal programs use a stack to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is executed and the required size is determined by the number and type of variables declared and the number of and sequence of procedure calls. Methods of estimating the amount of stack required for a program are included in a later section of this manual.

The runp command allows the amount of stack space to be specified on the command line. In the runp command, the size of the stack is selected by following the program name with the stack size, separated with a blank or a comma. For example, the following line would cause the program DATABASE to execute with 15K (15360 bytes) of stack space. (note that there are no angle brackets used with the runp command).

```
RUNP DATABASE 15K
```

The stack size can be specified as a decimal or hexadecimal number. Hexadecimal numbers have a '#' as the first character. This is the same notation as is used in the Pascal language. The letter 'K' means 1024, so 8K is equivalent to 8*1024 or 8192. If no stack size is specified, then one half of the unused memory space is allocated for the stack, and the other half to the heap. The heap is the area of memory used by the Pascal program for dynamic memory storage as required by the procedures NEW and DISPOSE.

When execution of the program completes, the amount of stack and heap used is displayed on the screen. These numbers reflect the actual quantity of memory used during execution.

THE PASCAL COMPILER LISTING

The Pascal compiler reads the source program from a file and produces two outputs. One of these is a file containing the object code. This code is loaded when the program is executed. The other output of the compiler is the listing. The listing contains the text of the source program with some additional information.

The listing is divided into pages. At the top of each page is a heading. The heading contains the version number of the compiler, and the page number. Each page after the first contains a form feed (control/L or #0C) character. The form feed will cause a page eject on most printers. The number of lines per page may be changed by a compiler option in the source program. See the Reference Manual.

Each line of the listing is numbered beginning with line 1. The compiler may also generate hexadecimal addresses for each line of the listing. The compiler widelist option causes this extra information to be generated. The addresses represent the locations of the generated object code relative to the start of the program. If the program contains procedures or functions, the addresses for these routines are relative to the start of the routine.

If errors are detected by the Pascal compiler, error messages will appear in the listing. Error message lines have a string of five asterisks ('*****') at the beginning of the line. An up arrow will appear pointing to the approximate location within the line where the error was detected. This will be followed by one or more error codes. It is possible for a single error to generate more than one error code. For example, a procedure argument which is an undefined variable also does not match the type of the parameter. In most cases, the first error code identifies the cause of the error.

If any errors are detected, a summary of the meanings of the error codes generated is printed at the end of the listing.

THE LINKLOAD COMMAND

This section describes the use of the Pascal linking loader. The linking loader provides powerful facilities for configuring Pascal programs. Separately compiled programs and procedures may be linked together and executed. Programs may be linked and stored as command files on disk and then later invoked from TRSDOS as commands. These command files behave in the same way as the utilities supplied with the operating system. This section assumes that the reader is familiar with the Reference Manual.

The loader is executed by typing LINKLOAD at the TRSDOS command level. At this point the linking loader is brought into memory from disk. The first item displayed is a menu of commands followed by the command prompt:

```
L=LOAD, R=RUN, T=TRSDOS, I=INIT, S=SYMBOLS, B=BUILD CMD
>>
```

Each of these commands will be described in detail later. All commands require only the single letter, although longer names will also be accepted. A command is terminated with the <enter> key. To execute a command, simply type its first letter followed by <enter>. If more information is required, additional prompts will be supplied. The list of commands can be displayed by typing H <enter> or ? <enter>.

L : The load command

The load command is used to load programs, procedures and functions into memory. To load a program, type "L" and press the <enter> key. The load command will ask for a file name. Type the name of the file in standard TRSDOS notation. The file should contain object code generated by the Pascal compiler. The object file will be opened and the object code will be loaded into memory. Each time a procedure or function is loaded, its name will be displayed on the screen. This will allow you to monitor the load process, and shows the identity of the modules being loaded. The program name will appear last.

The object code for each Pascal procedure is compiled into a separate entity. These are then linked together when they are loaded. This allows procedures to be compiled separately and then linked. Thus, a program may be compiled a piece at a time, and when changes are made, only the parts affected by the change need to be recompiled. This also allows the creation of libraries of utilities. These utilities can be loaded with any program that needs them, but need be compiled only once.

S : Symbols

The linking loader records the name and address of each procedure in a table as it is loaded. Also in this table are the names of procedures that have been called (referenced) by another procedure, but have not yet been loaded into memory. This symbol table can be displayed to the screen with the "S" command.

The symbols command displays all currently defined or referenced symbols on the screen. The display may be halted by pressing <shift> @ and continued by pressing <enter>. One procedure name is displayed per line. After the procedure name is a character that describes the use of that procedure. A "D" indicates that the name is defined; that is, the procedure has been loaded into memory. An "R" indicates that the procedure has been referenced but not yet defined. This means that a procedure that has already been loaded makes a call to this procedure. All procedures that are called must be loaded before the program can run. A "C" indicates that the symbol is the name of a common block. Commons are used to provide statically allocated shared data. See the Reference Manual for an explanation of the use of commons.

The last item on the line is the address of the symbol. If the symbol is defined ("D"), then this is the address in memory where the procedure begins. If the symbol has not been defined ("R"), then this is the address of the last place it was used (called).

R : RUN

After a program has been loaded, it can be executed with the Run command. The linking loader prompts for the amount of stack space required by the program. As in the RUNP program, the default is to allocate one-half of the unused memory to the stack, and the other half to the heap. If these space allocations are sufficient, then simply press the <enter> key. Otherwise enter a value. The size of the stack may be expressed in decimal, hexadecimal (precede the number with "#"), or in kilobytes. 8k means 8 times 1024, or 8192 bytes. Methods of estimating the required stack size are included in a later section of this manual.

The program will execute after the prompt is answered. If files are to be used in the program, the names of the files to be used will be determined from the keyboard. When a file is opened with RESET or REWRITE, the Pascal file name will be displayed on the screen and you will be requested to type the name of the actual file to be used. The names are in standard TRSDOS notation. You may also specify a device instead of a file name. The legal device names are :C (crt), :L (line printer), and :D (dummy). (Prompts for filenames may be eliminated by the use of the external procedure SETACNM. Files or device names may be built into the program with SETACNM. See the section on external procedures and functions in TRSLIB.

B : Build a command file

Once a program has been loaded, it may be saved on disk as a /CMD file. This is done by the build (B) command. The linking loader first prompts for the stack size as in the run command. The next prompt asks for a file name. This is the name of the file that will contain the program. The B command causes the program to be saved to disk in TRSDOS command file format and then exits to TRSDOS. The program may then be executed by simply typing the name of the command file from TRSDOS.

I : Init

The I command clears the symbol table and redisplays the command menu. This command may be used if the wrong file is loaded by mistake. It is equivalent to exiting to TRSDOS and then running LINKLOAD again.

T : TRSDOS

The T command returns to the TRSDOS operating system.

LINKLOAD ERROR MESSAGES

*** CANNOT OPEN FILE

This message is generated when the loader cannot find the file specified with the L command. This may be caused by a misspelling or the wrong disk being in the drive.

*** UNRESOLVED REFERENCES

When you use the run command to execute a program or the build command to generate an image on disk, the loader checks that all of the procedures that are called within the program have been loaded. If there are procedures or functions that have been called but have not been loaded, then this message is generated. At this point, you can load the required files and repeat the command. The S command may be used to list names of the procedures that are not yet defined. These will have an "R" in the listing.

*** INVALID OBJECT TAG

This message is displayed when a load is attempted on a file that is not a valid object file. The most frequent cause of this error is an attempt to load the source program instead of the object.

*** SYMBOL TABLE FULL

The linking loader has room for 256 different external symbols. If more procedures than this are loaded, the symbol table will become full.

*** ILLEGAL REFERENCE

This message signifies an inconsistent structure in an object file. It is an indication that the file has been damaged. The best solution is to recompile the offending program.

TRSDOS File Names and Device Names

In TRSDOS, a file name has four parts. The main file name is 1 to 8 characters long and identifies the file. This part must begin with a letter and may contain any alphanumeric characters.

The second part of the file name is an optional password. The password is a sequence of 1 to 8 alphanumeric characters, the first of which must be a letter. The password is used to limit access to file.

The third part of the file name is an optional extension which is separated from the rest of the name by a slash (/). The extension may be 1 to 3 characters in length and is usually used to identify the type of the file (eg. /PCL for Pascal source, /OBJ for object code, etc.).

TRS-80 Pascal uses certain file extensions as defaults. For example, when the Pascal compiler is executed with a file specified on the command line, the compiler assumes that the extension is PCL and it places the object code into a file with the same name but an extension of OBJ. The RUNP command assumes that its input has the extension OBJ unless otherwise specified.

OPTIMIZE by default takes its input from a file with an extension of OBJ and writes its output to a file by the same name with an extension of OPT. In a simlar manner, CODEGEN uses OBJ as a default input and COD as the corresponding output.

The fourth part of the file name is the disk drive. This part is also optional. The drive number is separated from the rest of the file name by a colon (:). If a drive number is not specified, all available drives will be searched. The search begins with drive number 0 and continues until the file is found or there are no more drives to search. If a drive number is specified, only that drive is searched. Specifying a drive number will insure that a file is placed on a specific drive and will also speed up file access time.

Example File Names:

```
ACCOUNT/DAT
SAMPLE2/OBJ:1
DATABASE
SECRET.REWQ/PCL
HOMEWRK:2
```

Device names are single characters preceded by a colon (:). The following devices are supported by the runtime.

:C	the CRT
:L	the Line Printer
:D	dummy device

ESTIMATING STACK SIZE

Pascal programs use a stack to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is run and the required size is determined by the number and type of variables declared and the number of and sequence of procedure calls. The stack is a dynamic structure. Space is allocated when a procedure is called and released when the procedure is exited.

The total stack size required by a program is determined from its dynamic behavior at run time. Each time a procedure is called, space is allocated for its local variables. The total stack in use is a function of the number of procedures active at the time and the number and sizes of variables used within those procedures. If two procedures are never active at the same time, then the space used by each can be shared. The total stack that must be allocated is determined from the maximum size that is in use at any given time.

The simplest method of determining stack requirements is to run the program. Specify enough stack for it to run, perhaps with an excess. When the program terminates, the maximum stack used by the program is printed on the CRT. A program may use differing amounts of stack each time it is executed. This often occurs when a program is driven by a users input. A good rule of thumb is to allocate 20% more stack than is required for a typical execution of the program.

The size of stack required can also be determined from the source program. It is necessary to determine which procedures will be active at a given time. Then add the size of the local variables for each procedure. If too little stack is allocated for the program, it may terminate with a runtime error.

The sizes of simple variables are summarized below:

type	size in bytes
CHAR	1
BOOLEAN	1
INTEGER	2
STRING	2
REAL	4
REAL (double precision)	8
FILE	32
TEXT	32

The size of an array is determined by multiplying the size of the array (upper bound-lower bound+1) by the size of an element. The size of a record is determined by adding the sizes of its individual fields. Packing is on byte boundaries.

The size of a set is one plus the ordinal of its largest possible member divided by 8. Enumerated types require one byte, and subranges require one or two bytes. (0..255 requires one byte, 0..256 requires two bytes).

To calculate the total stack size required, you should also include 64 bytes for the predeclared files INPUT and OUTPUT. Active procedures require space for their parameters as well as their local variables. Parameters passed by value require storage based on the size of the variable. Parameters passed by reference require two bytes each. Each active procedure also requires 9 bytes to store dynamic return information.

PASCAL MEMORY USAGE

The LINKLOAD and RUNP programs load at address Hex 3000. The object code for Pascal programs load immediately above the loader. The next segment above the program contains the Pascal stack. The stack contains local variables and return addresses from procedure and function calls.

The remainder of the available memory contains the heap. The heap is a section of memory that is used for dynamic storage. Programs that use pointers and the procedure NEW will use storage from the heap. The heap also contains the buffers used to read and write to files. The Pascal runtime support routines perform blocking on data from files. Each file is allocated a 256 byte buffer from the heap and information is read or written to this buffer before being transferred to disk. This improves performance by decreasing the frequency of disk accesses.

COMPILER MEMORY CONSTRAINTS

The minimum stack required by PASCAL/CMD is currently 3900 bytes. If you specify less than this amount, 3900 will be used. PASCAL/CMD requires this much stack even though the stack used message at the end of a compile may indicate that less was needed. PASCALB/CMD has no such minimum.

PASCAL/CMD allows enough space for approximately 500 symbols to exist in a program being compiled. Each identifier used in a program requires an entry in the compilers symbol table. Named constants, types, variables, procedure and function names are all identifiers in Pascal programs and are entered into the compilers symbol table.

There are some ways of saving memory during the compile so that larger programs can be compiled. The limit on symbols is relative to the number of symbols visible at any point within the program. Symbols that are not available to the program are not retained by the compiler. The use of symbol table space can be improved by defining fewer global variables at the outer levels and making use of locals whenever possible. This is also good programming practice.

The length of symbol names is not relevant in Pascal, unlike BASIC. Use of long names has no effect on program size or compiler memory usage. However, extensive use of string and real constants will cause the compiler to use more memory.

PASCALB is the overlayed or segmented version of the compiler. This version dynamically loads portions of the Pascal compiler from disk as needed. This increases the amount of memory available for symbols and allows larger programs to be compiled. The overlayed compiler will compile larger programs. I.E.; a typical 6000 line program will compile successfully on a Model 4 system. The overlayed compiler will run more slowly due to overlay loading. Therefore, use the non-overlayed compiler until memory becomes a problem.

REAL NUMBERS

Real numbers are either single precision or double precision. Whether real numbers in TRS-80 Pascal are considered to be single or double precision is set by an option at compile time.
See the appendix of the Reference manual for a description of the DOUBLE compiler option.

	Accuracy	Range
Single precision	6 digits	(-)1.7E-38..(-)1.7E+38
Double precision	16 digits	(-)1.7E-38..(-)1.7E+38

Note: All trancendental functions are performed in double precision whether single or double is specified. This is to avoid round off errors that lower the accuracy of the result. The functions are calculated to 9 digits accuracy. Single precision numbers are rounded before truncation.

TRS-80 Pascal is supported by 3 libraries of procedures and functions (TRSLIB/OBJ, RANDOM/OBJ, and STRINGS/OBJ). TRSLIB/OBJ contains routines which provide access to specific Model 4 or TRSDOS Version 6 features. RANDOM/OBJ contains random access file routines. STRINGS/OBJ contains a set of dynamic string functions. A routine from one of these libraries may be used by a Pascal program simply by declaring the routine as an EXTERNAL procedure or function.

Pascal programs may be executed with either the RUNP or the LINKLOAD program. The RUNP program contains all the routines in the 3 libraries. When executing a program with RUNP, any library routine which is externally declared is automatically linked into the program. The LINKLOAD program does not contain any of the routines in the 3 libraries. When executing a program with LINKLOAD, any library routine which is externally declared must be linked to the program by loading the library file which contains the routine.

Each of the library routines is described in the following pages. A Pascal external declaration is given for each routine. This declaration should be included in any program that uses the routine. The external declarations for the library routines are included in files TRSLIB/PCL, RANDOM/PCL, and STRINGS/PCL. Any or all of these declarations can be inserted into a Pascal program using the INSFILE command of the text editor.

In the following descriptions the type byte is frequently used. This type may be declared as: BYTE = 0..255 . When a variable of type byte is used, it will occupy a single byte of storage.

SVC

```
PROCEDURE SVC(VAR A, STATUS: BYTE;
              VAR BC, DE, HL, IX, IY: INTEGER); EXTERNAL;
```

SVC is used to make TRSDOS Version 6 supervisor calls. Supervisor calls provide the mechanism for executing various TRSDOS operating system routines. See the Technical Reference Manual (Cat. No. 26-2110) for an explanation of the available supervisor calls.

The parameters passed to SVC will be loaded into the Z-80 registers. The parameters will also return the values of the Z-80 registers when the SVC routine terminates. The A register is used to specify an SVC number which determines which operating system routine is executed. Each operating system routine has specifications for which Z-80 registers are used to pass information.

TIME

```
TYPE ALPHA = PACKED ARRAY[1..8] OF CHAR;
PROCEDURE TIME(VAR T: ALPHA); EXTERNAL;
```

TIME returns the current time of the system clock in the form hh:mm:ss.

DATE

```
TYPE ALPHA = PACKED ARRAY[1..8] OF CHAR;
PROCEDURE DATE(VAR T: ALPHA); EXTERNAL;
```

DATE returns the current date of the system clock in the form mm/dd/yy.

SOUND

```
PROCEDURE SOUND(TONE, DURATION : INTEGER); EXTERNAL;
```

SOUND is used to generate sound using specified tone and duration codes. The TONE parameter should be passed as a number between 0 and 7 with 0 being the highest tone and 7 being the lowest. The DURATION parameter should be passed as a number between 0 and 31 with 0 being the shortest and 31 being the longest.

CMDLINE

```
TYPE STRINGPTR  = ^CHARSTRING;
     CHARSTRING = PACKED ARRAY[1..80] OF CHAR;

PROCEDURE CMDLINE(VAR LOCATION, ORIGIN : STRINGPTR); EXTERNAL;
```

The CMDLINE procedure returns pointers to the command line stored by the operating system. Each time a command is executed from the TRSDOS Ready prompt, all characters typed are stored in a buffer within the operating system. For example, when RUNP DATABASE FILE1 <enter> is typed, the operating system stores RUNP DATABASE FILE1 in the buffer.

The ORIGIN parameter returns a pointer to an array which contains the entire command line buffer. The first element of the array is the first character of the command line. The LOCATION parameter returns a pointer to an array which contains only the part of the buffer which begins with the first non-blank character following the command name.

Using the above command line as an example,

```
ORIGIN^       = RUNP DATABASE FILE1
ORIGIN^[1]    = R
LOCATION^     = DATABASE FILE1
LOCATION^[1] = D
```

USER

```
PROCEDURE USER(ADDRESS : INTEGER; VAR DATA : INTEGER); EXTERNAL;
```

This procedure interfaces to assembly language routines resident in memory. ADDRESS is the physical address where the routine is loaded.

Information is passed to the assembly language routine through the DATA parameter. When the assembly language routine is called, the HL register pair contains the value of DATA. When the routine exits, the contents of the HL register pair is returned as the new value of DATA. In cases where more than one word of information is required, the value of DATA can be the address of a variable. The address of any Pascal variable can be obtained using the predefined LOCATION function (eg. addr := LOCATION(x)).

The assembly language routine is entered with a standard Z80 call instruction and should be exited via a return. All Z80 registers are available for use in the assembly language subroutine.

CALL$

```
PROCEDURE CALL$(ADDRESS : INTEGER; VAR A,STATUS: BYTE;
          VAR BC, DE, HL, IX, IY : INTEGER); EXTERNAL;
```

This procedure can be used in a similar manner to USER to call assembly language subroutines. The difference is that CALL$ permits you to set up all of the Z80 registers from Pascal. The values passed (except status) will be in the registers when the subroutine is called. When the subroutine returns, the current contents of all registers are returned to the Pascal program via the reference parameters. Status is the Z-80 flag register.

$MEMORY

```
PROCEDURE $MEMORY(VAR STACK, HEAP : INTEGER); EXTERNAL;
```

This procedure allows a program to determine the amount of memory currently available. The parameter STACK returns the current number of stack bytes available and the parameter HEAP returns the amount of heap available.

HP$ERROR

```
PROCEDURE HP$ERROR(NEWSTATE : BOOLEAN;
          VAR OLDSTATE : BOOLEAN); EXTERNAL;
```

This procedure sets the state of the heap error recovery flag within the Pascal runtime system. When this flag is set to true, then a call to the procedure NEW will cause the program to terminate with an error message if no more space is available. Setting this flag to false causes the procedure NEW to return NIL if no space is available. The calling program should check for NIL on each call to NEW when this flag is set to false. This allows a program to use maximum memory from the heap without danger of an abnormal termination when space is exhausted.

PEEK

```
FUNCTION PEEK(ADDRESS : INTEGER) : BYTE; EXTERNAL;
```

This function returns the contents of any memory location. It may be used to examine memory or memory mapped input devices. ADDRESS is the address being examined. An address may be passed if its value is known. The addresses of Pascal variables may be obtained by calling the LOCATION function. The LOCATION function is a predeclared function which is described in chapter 9 of the Reference Manual.

POKE

```
PROCEDURE POKE(ADDRESS : INTEGER; VALUE : BYTE); EXTERNAL;
```

Poke is used to alter the contents of any location in memory. It may also be used to write to memory mapped output devices.

INP
===

```
FUNCTION INP(PORT : BYTE) : BYTE; EXTERNAL;
```

This function performs input from a Z80 IO port. The port number is passed to the function and the value read from that port is returned as the function value.

OUT
===

```
PROCEDURE OUT(PORT, VALUE : BYTE); EXTERNAL;
```

This procedure performs physical output to a Z80 port. It may be used in conjunction with the function INP to communicate with devices interfaced as input or output ports. The two parameters specify the port number and the value to be written to that port.

WRITECH
=======

```
PROCEDURE WRITECH(CH : CHAR); EXTERNAL;
```

This procedure writes a single character to the terminal.

WRITESTRING
===========

```
TYPE
   CHARSTRING = PACKED ARRAY[1..XX] OF CHAR;
PROCEDURE WRITESTRING(VAR S : CHARSTRING; FIRST, LAST : INTEGER);
          EXTERNAL;    (XX is any length)
```

This procedure writes a portion of a string of characters to the terminal. FIRST is the index of the first character to be written, LAST is the index of the last character to be written. The total number of characters displayed is: LAST-FIRST+1. If last is less than first then no characters are written. The type CHARSTRING may be declared as a packed array of any length convenient for the application. The XX above should be replaced by this value, or XX should be declared as a constant.

INKEY
=====

```
PROCEDURE INKEY(VAR CH : CHAR; VAR READY : BOOLEAN); EXTERNAL;
```

This procedure attempts to obtain a character from the keyboard. If a character is available, then CH is the character and READY is set to TRUE. If no key is pressed, then READY is FALSE and CH is the space character: ' '.

GETKEY
======

```
FUNCTION GETKEY : CHAR; EXTERNAL;
```

This function waits for and returns the next character from the keyboard.

File Routines

FILE$STATUS
===========

```
FUNCTION FILE$STATUS(VAR F : TEXT) : BYTE; EXTERNAL;
```

This function returns the status of a file. The file can be of any type, but the external declaration must specify a type that matches the type of file being tested. The byte returned is the operating system error code for the latest IO (input or output) error. If no errors have occurred, then zero is returned. This function is used in conjunction with IO$ERROR and allows a program to detect and recover from its own IO errors.

IO$ERROR

```
PROCEDURE IO$ERROR(NEWSTATE : BOOLEAN;
          VAR OLDSTATE : BOOLEAN); EXTERNAL;
```

This procedure sets the state of the IO error recovery flag within the Pascal runtime system. This flag is used to determine whether a program detects its own IO errors. If the flag is set to true, then default error processing is performed. In case of an error on a file or device, a message is displayed on the CRT and the program halts.

If the IO error flag is set to false, then all IO errors are ignored by the system, and it is up to the program to check for and recover from IO errors. IO errors can be detected by calling the function FILE$STATUS. NEWSTATE is a boolean value that sets the new state of the IO error recovery flag. OLDSTATE is used to return the previous value of the flag. This allows a program to change the state temporarily and then restore it.

DELFILE

```
TYPE PATH = PACKED ARRAY[1..xx] OF CHAR; {xx is any length}
PROCEDURE DELFILE(VAR FILENM : PATH; VAR STATUS : INTEGER);
          EXTERNAL;
```

This procedure deletes a file from any disk in the system. FILENM should be the TRSDOS name of the file, including (optional) drive specification. The file name should be terminated by a carriage return (#0D). STATUS is 0 if the operation is successful. A status of other than 0 is an operating system error code number.

RENAME

```
TYPE PATH = PACKED ARRAY[1..xx] OF CHAR;  {xx is any length}
PROCEDURE RENAME(VAR OLDNAME, NEWNAME : PATH;
          VAR STATUS : INTEGER); EXTERNAL;
```

RENAME changes the name of a TRSDOS file. The OLDNAME and NEWNAME should be the file names as in DELFILE (above). Both file names must map to the same drive and there must not be a file on that disk with NEWNAME as its name (use DELFILE first if necessary). The file must not be open by the program when rename is called. Status is 0 if the operation is successful.

SET$ACNM

Note: The SETACNM procedure described on the following page performs the same function as SET$ACNM and is easier to use.

```
TYPE
   FILENM = PACKED ARRAY[1..XX] OF CHAR;
   ALPHA  = PACKED ARRAY[1..8] OF CHAR;
     (Where XX is any length long enough for the filename)
 PROCEDURE SET$ACNM(VAR F : TEXT; VAR file name : FILENM;
         NAMELENGTH : INTEGER; VAR FILEID : ALPHA); EXTERNAL;
```

SET$ACNM is used to set the name of the physical file or device to be associated with a Pascal file. It allows a program to compute file names internally. For example, a database program may know the name of the file containing the database. This procedure allows the program to specify the file name rather than requesting it from the keyboard.

The parameter F can be a file of any type. The external declaration of SET$ACNM that is included in the source program must specify a type for F that matches the actual file type to be used. File name is a string containing the text of the file name. This string must be compatible with the operating system syntax for file names. NAMELENGTH is an integer that specifies the length of the file name. FILEID is an 8 character string that is used to identify the Pascal name for the file, such as INPUT or OUTPUT. The first character of fileid must be an uppercase letter.

If SET$ACNM is called prior to a RESET or REWRITE on a file, then Pascal will not prompt the CRT for the file name. All subsequent RESET or REWRITES will not cause a prompt unless a CLOSE(file name) is performed on the file. The file name association will remain as previously defined by SET$ACNM.

(Example program segment)

```
TYPE
FILENAME = PACKED ARRAY [1..15] OF CHAR;
ALPHA    = PACKED ARRAY [1..8] OF CHAR;
VAR  FNAME : FILENAME;
     FILEID: ALPHA;
     F     : TEXT;
PROCEDURE SET$ACNM(VAR F:TEXT; VAR FNAME:FILENAME; LEN:INTEGER;
                   VAR FILEID:ALPHA); EXTERNAL;
BEGIN
   (* this assignment statement requires the name to be left *)
   (* justified, and blank padded to the correct array length *)
   FNAME:='DATA/TXT:0     ';
   FILEID:='F       ';
   SET$ACNM(F,FNAME,10,FILEID);
   RESET(F);
   READ(F,CH);
   (* AND ETC..... *)
```

SETACNM

```
PROCEDURE SETACNM(VAR logical : filetype;
          physical : STRING); EXTERNAL;
```

The library procedure SETACNM serves the same purpose as SET$ACNM but is simpler to use. The procedure takes only two parameters, the Pascal logical file variable, and the physical file or device name to associate with it. Filetype is any legal Pascal file type. The physical name parameter is a dynamic string. The SETACNM procedure disposes this string before exiting, to recover the space.

If multiple file types are used in a program, the type transfer operator (::) may be used to allow SETACNM to be called with different file types. The external declaration of SETACNM may specify one of the file types used. The type transfer operator must then be used with the other file types to avoid a type mismatch error during the compile. Each of the other files must be type transferred to the same type as the one used in the declaration. The following example illustrates the use of SETACNM.

```
(*$NO INOUT*) {eliminate the prompt for INPUT & OUTPUT}
PROGRAM sample;

VAR printer : TEXT;
    out     : FILE OF INTEGER;

PROCEDURE SETACNM(VAR f : TEXT; name : STRING); EXTERNAL;

BEGIN          {main body of program sample}
  {map logical file "printer" to the line printer}
  SETACNM(printer,BLDSTR(':L'));
  {no prompt will occur when REWRITE(printer) is executed}
  REWRITE(printer);
  {map logical file "out" to disk file "OUT/DAT"}
  SETACNM(out::TEXT,BLDSTR('OUT/DAT'));
  {no prompt will occur when REWRITE(out) is executed}
  REWRITE(out);
  ...
  ...
 END.          {end of program sample}
```

CLEARGRAPHICS

```
PROCEDURE CLEARGRAPHICS; EXTERNAL;
```

This procedure clears the screen with blanks.

CLEARSCREEN

```
PROCEDURE CLEARSCREEN; EXTERNAL;
```

This procedure does the same thing as CLEARGRAPHICS.

GOTOXY

```
PROCEDURE GOTOXY(X, Y : INTEGER); EXTERNAL;
```

This procedure positions the cursor on the screen to the specified location. The value of X should be in the range of 0 to 79 and the value of Y should be in the range of 0 to 23. The top left corner of the screen corresponds to X = 0 and Y = 0.

NOBLANK

```
PROCEDURE NOBLANK(REDISPLAY : BOOLEAN); EXTERNAL;
```

When the Model 4 video screen receives a carriage return (#0D), the next line after the line containing the cursor is erased. The NOBLANK procedure is used in conjunction with input files which are connected to the keyboard to prevent the next line from being erased when the <enter> key is pressed. The NOBLANK procedure must be called with the parameter REDISPLAY set to TRUE to prevent the <enter> key from erasing the next line. NOBLANK must be called before the input file is opened (RESET) in order to have any effect. Therefore, the predeclared file INPUT cannot be used unless the (*$NO INOUT) compiler option is used to prevent it from being automatically opened.

When a program is executed from a JCL file, an input file connected to the keyboard receives input from the JCL file instead. To prevent this from occurring, NOBLANK(TRUE) may be executed prior to opening the input file.

READCURSOR

```
PROCEDURE READCURSOR(VAR X, Y : INTEGER); EXTERNAL;
```

This procedure returns the current position of the cursor on the screen. X is in the range of 0 to 79 and Y is in the range of 0 to 23.

RSETPOINT

```
PROCEDURE RSETPOINT(X, Y : INTEGER); EXTERNAL;
```

This procedure clears (turns off) a graphics point on the screen. The location of the point is specified by the X and Y parameters. X should be in the range of 0 to 159 and Y should be in the range of 0 to 71.

SETPOINT

```
PROCEDURE SETPOINT(X, Y : INTEGER); EXTERNAL;
```

This procedure sets (turns on) a graphics point on the screen. The location of the point is specified with the X and Y parameters. X should be in the range of 0 to 159 and Y should be in the range of 0 to 71.

TESTPOINT

```
FUNCTION TESTPOINT(X, Y : INTEGER) : BOOLEAN; EXTERNAL;
```

This function tests the state of a graphics point on the screen. The location of the point is specified with the X and Y parameters. X should be in the range of 0 to 159 and Y should be in the range of 0 to 71. The value returned for the function is TRUE if the point is set (turned on) and FALSE if the point is cleared (turned off).

EXTMEM

```
TYPE MEMOPCODE = (M_RELEASE, M_TEST, M_RESERVE, M_GET, M_PUT);
PROCEDURE EXTMEM(OPERATION : MEMOPCODE; BANK : INTEGER;
                 LOCALADDRESS, EXTENDADDRESS, BLOCKSIZE : INTEGER;
                 VAR STATUS : INTEGER); EXTERNAL;
```

The TRS-80 model 4 can contain up to 128k bytes of memory. This procedure allows a Pascal program to use the top 64k of memory to store data under program control. For this procedure to work, at least one bank of 32k must be free (not used by memdisk or some other program). The parameter BANK is used to specify the bank number in extended memory. The two upper banks in a 128k machine are banks 1 and 2.

The operation code tells EXTMEM which operation to perform. The EXTMEM procedure supplies all needed operations including allocating and releasing banks of memory. In each case the variable STATUS contains the result code when EXTMEM returns. If STATUS is 0, the operation completed successfully, otherwise it is the returned status code from the operating system. See the TRSDOS 6 Technical Manual for details.

M_RELEASE
  The bank of memory is released.

M_TEST
  The bank is tested to determine its current state. If status is 1 then the bank is busy (in use) and if status is 0 then the bank is available.

M_RESERVE
  Reserves the selected bank of memory and makes it available for use by EXTMEM. The selected bank is marked as being in use.

M_GET
  Copys a block of data from extended memory to local memory. EXTENDADDRESS is the address of the block in extended memory. The addresses in extended memory range from #8000 to #FFFF. LOCALADDRESS is the address for the block in local memory. This address can be obtained by use of the LOCATION function in Pascal. BLOCKSIZE is the size of the block in bytes. The size of a data structure can be obtained using the SIZE function in Pascal.

M_PUT
  Copys a block of data from local memory to extended memory. The parameters are the same as for M_GET.

The following sample program illustrates use of EXTMEM. An array is stored and retrieved from extended memory.

```
PROGRAM SHOWEXTMEM;
TYPE
  MEMOPCODE = (M_RELEASE, M_TEST, M_RESERVE, M_GET, M_PUT);
  REAL_ARRAY = ARRAY[0..120] OF REAL;
VAR
  R      : REAL_ARRAY;
  STATUS : INTEGER;

PROCEDURE EXTMEM(OPERATION : MEMOPCODE; BANK : INTEGER;
          LOCALADDRESS, EXTENDADDRESS, BLOCKSIZE : INTEGER;
          VAR STATUS : INTEGER); EXTERNAL;

BEGIN
  { allocate bank 1 of extended memory }
  EXTMEM(M_RESERVE,1,LOCATION(R),#8000,SIZE(REAL_ARRAY),STATUS)
  IF STATUS <> 0 THEN WRITELN('unable to allocate bank 1')
  ELSE BEGIN
    { some code to enter data into the array should go here }

    { store data in extended memory }
    EXTMEM(M_PUT,1,LOCATION(R),#8000,SIZE(REAL_ARRAY),STATUS);
    IF STATUS <> 0 THEN BEGIN
      WRITELN('can''t store data in memory');
      ESCAPE;
      END;

    { retrieve the data }
    EXTMEM(M_GET,1,LOCATION(R),#8000,SIZE(REAL_ARRAY),STATUS);
    IF STATUS <> 0 THEN BEGIN
      WRITELN('can''t get data from extended memory');
      ESCAPE;
      END;

    EXTMEM(M_RELEASE,1,LOCATION(R),#8000,SIZE(REAL_ARRAY),STATUS);
    END;
  WRITELN('test completed');
END.
```

Standard Pascal defines a file as a sequence and allows files to be read or written only from beginning to end in sequential order. Random Access files are files in which records can be both read and written in any order.

The following Pascal procedures and functions are provided to allow random access to records in a file. When using random access files, these routines should be declared as external. The following external declarations for the random file routines use the following types.

filetype - A user defined type of the form:
filetype = FILE OF datatype
datatype - A predefined type such as INTEGER or a user defined type such as datatype = RECORD f1 : INTEGER; f2 : REAL END; The number of bytes in datatype must be between 1 and 256.

OPENRAND

```
PROCEDURE OPENRAND(VAR f:filetype; recordlen:INTEGER; pathname:STRING;
                   VAR status:INTEGER); EXTERNAL;
```

A random file must be opened prior to a read or write operation. The OPENRAND procedure opens a random access file.

f - The logical Pascal name for the random file.
recordlen - The length in bytes of datatype. The size of datatype may be determined using the SIZE function. For example, recordlen:=SIZE(INTEGER) or recordlen:=SIZE(datatype). The value of recordlen must be between 1 and 256.
pathname - The physical disk file name for the random file. For example, pathname:=BLDSTR('DATABASE/DAT').
status - The returned error code status. A returned status of 0 indicates that the open was successful. Otherwise, there was an error in attempting to open the file.

READRAND

```
PROCEDURE READRAND(VAR f:filetype; recordnum:INTEGER;
          VAR dat:datatype; VAR status:INTEGER); EXTERNAL;
```

The READRAND procedure reads a record from a random file.

f - The logical Pascal name for the file.
recordnum - The random file record number. Recordnum must be between 0 and 32767.
dat - The variable that contains the data read from the file.
status - The returned error code status. A returned status of 0 indicates that the read was successful. Otherwise, there was an error in attempting to read from the file.

WRITERAND

```
PROCEDURE WRITERAND(VAR f:filetype; recordunm:INTEGER;
          VAR dat:datatype; VAR status:INTEGER); EXTERNAL;
```

The WRITERAND procedure writes a record to a random file.

f	- The logical Pascal name for the file.
recordnum	- The random file record number. Recordnum must be between 0 and 32767.
dat	- The variable which contains the data written to the file.
status	- The returned error code status. A returned status of 0 indicates that the write was successful. Otherwise, there was an error in attempting to write to the file.

CLOSERAND

```
PROCEDURE CLOSERAND(VAR f:filetype); EXTERNAL;
```

All random files must be explicitly closed.
The CLOSERAND procedure closes a random file.

f	- The logical Pascal name for the file.

As with random files on any operating system, there are some peculiarities about random files. For example:

(1) If you WRITE record number 1 and WRITE record number 100, and then read any record from 2 to 99, the returned buffer will contain trash. The data will be whatever was previously on the diskette, probably the contents of an old file. This is because the operating system does not keep that much context. It is up to the user to keep track of unwritten records so they are not read.

(2) All blocking is taken care of by the system.

(3) The standard functions EOLN, EOF have no meaning for random files. The status codes as returned by the above routines perform those functions where applicable.

(4) The procedure OPENRAND is used to open a file for reading and writing. Opening an empty file and reading is perfectly legal.

(5) Random file record numbers are defined from 0..32,767 .

(6) As with normal files, if a file is declared locally within a procedure (ie. not passed in by reference) and opened, once the procedure is exited, Pascal will automatically close the file using the standard CLOSE file routine for non-random files and position the EOF mark in the directory at the last record read or written. This may not be the correct position as desired by the programmer. An explicit call to CLOSERAND should be used to close the random file and position the EOF. This will always correctly place the EOF mark.

(7) You may declare a file to be:

```
(*WHERE XX IS ANY RECORD LENGTH FROM 1 TO 32,767*)
TYPE  LINE = ARRAY(.1..XX.) OF CHAR;
VAR   F:FILE OF LINE;
```

Once the file has been opened, you may access it by using the READRAND and WRITERAND external procedures even if the file was not created by Pascal. There is only one procedure for opening random files (no reset and rewrite). You may read or write to a random file.

Random File Error Codes
Returned By Status Parameter

128 - PATH NAME IS NULL OR TOO LONG
129 - RECORD LENGTH TO LARGE
130 - FILE IS ALREADY OPEN
131 - FILE IS NOT OPEN

Any other returned code is an operating system code.
(See the Model 4 Disk System Owner's Manual)

If multiple random file types are used in a program, the type transfer operator (::) may be used to allow the random file routines to be called with different file and data types. The declarations may specify one of the file and data types used in the program. Any other files used must then utilize the type transfer operator when calling one of the random file routines. The other file and data types must be type transferred to the same types used in the declarations to avoid a type mismatch error during the compile. The following example illustrates the use of the random file routines. The status may be checked after each random file operation to determine if an error occurred. The returned status will be 0 if no error is detected during an operation.

```
PROGRAM sample;
TYPE filel = FILE OF CHAR;
     file2 = FILE OF INTEGER;
VAR  fl    : filel;
     f2    : file2;
     valuel, ch : CHAR;
     value2, status, number : INTEGER;
  PROCEDURE OPENRAND(VAR f : filel; length : INTEGER;
        name : STRING; VAR status : INTEGER); EXTERNAL;
  PROCEDURE CLOSERAND(VAR f : filel); EXTERNAL;
  PROCEDURE READRAND(VAR f : filel; number : INTEGER;
        VAR data : CHAR; VAR status : INTEGER); EXTERNAL;
  PROCEDURE WRITERAND(VAR f: filel; number : INTEGER;
        VAR data : CHAR; VAR status : INTEGER); EXTERNAL;
  PROCEDURE checkstatus(status : INTEGER);
  BEGIN
    IF status<>0 THEN
      WRITELN('* I/O ERROR: code number = ',status:3,' *')
  END;
BEGIN
  {open file "Fl/DAT"}
  OPENRAND(fl,SIZE(CHAR),BLDSTR('Fl/DAT'),status);
  {open file "F2/DAT"}
  OPENRAND(f2::filel,SIZE(INTEGER),BLDSTR('F2/DAT'),status);
  FOR number := 0 TO 255 DO
    BEGIN
    {write the ascii character set to Fl/DAT}
    ch := CHR(number);
    WRITERAND(fl,number,ch,status);
    {write the ordinal values of the character set to F2/DAT}
    WRITERAND(f2::filel,number,number::CHAR,status);
    END;
  FOR number := 0 TO 255 DO
    BEGIN
    {read the ascii character set from Fl/DAT}
    READRAND(fl,number,valuel,status);
    {read the ordinal values of the character set from F2/DAT}
    READRAND(f2::filel,number,value2::CHAR,status);
    END;
  checkstatus(status);     {check error status}
  CLOSERAND(fl);           {close Fl/DAT}
  CLOSERAND(f2::filel)     {close F2/DAT}
END.
```

The following functions are provided for handling dynamic string manipulations.
(See the appendix of the Reference Manual for information about the type STRING).

```
FUNCTION LEN(S : STRING) : INTEGER;
```
This function returns the length of a string.

```
FUNCTION LEFT$(S : STRING; POSITION : INTEGER) : STRING;
```
This function returns the left portion of the string ending at the specified position within the string.

```
FUNCTION RIGHT$(S : STRING; POSITION : INTEGER) : STRING;
```
This function returns the right portion of the string starting at the specified position within the string.

```
FUNCTION MID$(S : STRING; POSITION, LENGTH : INTEGER) : STRING;
```
This function returns the portion of the string starting at the specified position and including the number of characters specified by length.

```
FUNCTION STR$(LENGTH : INTEGER; CH : CHAR) : STRING;
```
This function returns a string of the specified length which is filled with the specified character.

```
FUNCTION ENCODEI(N : INTEGER) : STRING;
```
This function returns a string which is the character representation of the specified integer.

```
FUNCTION ENCODER(R : REAL) : STRING;
```
This function returns a string which is the character representation of the specified real. For single precision.

```
FUNCTION ENCODED(R : REAL) : STRING;
```
Same as ENCODER, but for double precision reals.

```
FUNCTION DECODEI(S : STRING) : INTEGER;
```
This function returns an integer number which is the binary representation of the specified string.

```
FUNCTION DECODER(S : STRING) : REAL;
```
This function returns a real number which is the binary representation of the specified string. For single precision.

```
FUNCTION DECODED(S : STRING) : REAL;
```
Same as DECODER, but for double precision reals.

```
FUNCTION CHARACTER(S : STRING; POSITION : INTEGER) : CHAR;
```
This function returns the character at the specified position in the string.

```
TYPE COMPAREVALUE = (LESS, EQUAL, GREATER);
FUNCTION CMPSTR(S1, S2 : STRING) : COMPAREVALUE;
```
This function compares the two specified strings and returns an enumerated value based on the comparison. The returned value is LESS if S1<S2, EQUAL if S1=S2, and GREATER if S1>S2.

```
FUNCTION CONC(S1, S2 : STRING) : STRING;
```
This function returns a string which is the result of the concatenation of the two specified strings.

```
FUNCTION CPYSTR(S : STRING) : STRING;
```
This function returns a copy of the specified string. The typical use for this function is in the assignment of one string variable to another. This prevents both string variables from referencing the same string. EG. STRING1:=CPYSTR(STRING2); will cause STRING1 to refer to a different copy of STRING2. STRING1:=STRING2; causes STRING1 to refer to the same copy of STRING2 and any changes in the value of STRING1 would cause STRING2 to change also.

```
FUNCTION DELETE(S : STRING; POSITION, LENGTH : INTEGER) : STRING;
```
This function returns the string which results after deleting a specified number of characters beginning at the specified position in the string.

```
FUNCTION FIND(SUBSTRING, S : STRING) : INTEGER;
```
This function returns an integer number which points to the start of the specified substring within the specified string. If the string does not contain the substring then the returned value is 0.

```
FUNCTION INSERT(SUBSTRING, S : STRING; POSITION : INTEGER) : STRING;
```
This function returns a string which is the result of inserting the specified substring into the specified string at the specified position.

```
FUNCTION REPLACE(OLDSTRING, NEWSTRING , S : STRING) : STRING;
```
This function returns the string which results after replacing the old substring with a new substring within string S.

Linking Assembly Language to Pascal

The LINKLOAD program may be used to link assembly language subroutines to Pascal programs. Any available assembler may be used to assemble the assembly language subroutine.

A Pascal program which uses an assembly language subroutine should declare an external procedure to represent the assembly language module. Choose a name for the procedure and declare it with no parameters. (eg. PROCEDURE GRAPH; EXTERNAL; could be used if you are linking to a subroutine to do graphics). The CALL$ library procedure should be used to call the assembly language subroutine. The first parameter to CALL$ is the address of the assembly language subroutine. Use the LOCATION function to obtain the address of the externally declared procedure (eg. address:=LOCATION(GRAPH)). The remaining parameters of CALL$ allow you to define values for the Z-80 registers (See CALL$ in the TRSLIB library of routines). The Pascal program should define values for the registers that the subroutine uses as input parameters. All register values are returned to the Pascal program through the parameter list of CALL$ when a return instruction is executed.

First compile the Pascal program and then follow these steps.

step 1) Origin the assembly language routine to load at hexadecimal address #70FE and assemble it. Note the size (in bytes) of the assembled routine.

step 2) Using the text editor, create a file containing the following two lines.

```
P0075E
GxxxxyyyyyyyyPzzzzE
```

where

xxxx	is the hexadecimal offset (in bytes) of the entry point from the assembled origin. If the origin is the entry point for the routine, then xxxx is 0000.
yyyyyyyy	is the name of the external procedure declared in the Pascal program. The name must be 8 characters long. If the name used in the Pascal program is shorter, left justify the name and pad with blanks. If the name is longer, use the first 8 characters only.
zzzz	is the hexadecimal size (in bytes) of the assembled subroutine.

step 3) Load the assembled subroutine into memory. This may be done with the operating system LOAD command if the assembler creates /CMD files.

step 4) Execute the LINKLOAD program and first load in the file created in step 2.

step 5) Next load in the Pascal program modules.

step 6) Run the program or Build a command file.

Multiple assembly language subroutines may be linked to a Pascal program. Simply make an external declaration for each subroutine in the Pascal program and use the LOCATION function to define the address parameters for CALL$.

Origin the first subroutine to load at address #70FE and assemble it. Add the size of the assembled routine to #70FE and use this as the origin for the second subroutine. Assemble the second subroutine. Add the size of the second assembled subroutine to its origin and use this as the origin for the third subroutine. etc...

Next create a file containing P0075E followed by the names of the external procedures declared for each subroutine and the sizes of each assembled routine. The routines must be listed in the order that they were assembled.

```
P0075E
GxxxxyyyyyyyyPzzzzE          "assembled first"
     .
     .
     .
GxxxxyyyyyyyyPzzzzE          "assembled last"
```

Use the operating system LOAD command to load all of the assembled routines.

Execute the LINKLOAD program and load in the file created above, followed by the Pascal object modules and either run the program or build an executable command file.

Generating EOF from the Keyboard

When performing input from the keyboard, EOF can be generated by the backquote character (`). On the Model 4, backquote is generated by SHIFT @.

PATCHES

Normally, when a program built with the linking loader terminates, the ending address and stack and heap used are sent to the screen. After a program has been completely debugged, this information is not needed. The following patch will eliminate these messages. The patch should be applied to a copy of the linking loader. After the patch is applied, any command file built with the patched copy of the linking loader will not print the stack and heap message.

1. Make a copy of the linking loader.

```
TRSDOS Ready
COPY LINKLOAD/CMD TO LINK/CMD
```

2. Apply the patch using the TRSDOS PATCH command.

```
TRSDOS Ready
PATCH LINK/CMD (X'6A98'=C3 13 6B)
```

TUTORIAL

Table of Contents

Chapter		Page
1	Introduction ..... Historical of Pascal. Why Pascal ?	2
2	Starting Concepts ..... Program, begin, end, write, writeln.	6
3	Data Concepts ..... Variables,Types: integer, real, char, text and boolean. Const section.	9
4	Advanced I/O ..... Rewrite, write, writeln, reset, read, readln.	13
5	Statements ..... Assignment statements, compound statements, multiply, divide, add, subtract.	17
6	Flow control ..... For loop and case statement.	23
7	Decision testing ..... Logical operators AND, OR, NOT. Relational operators > , < , >= , <= , <> , = . Flow control: IF, WHILE, REPEAT loop control statements.	27
8	Procedures and functions ..... Global and local variables, parameters, scoping, nesting.	32
9	Advanced data types ..... Structured data type: The array, record. With statements. User defined types: Enumerated, subrange. File of TYPE.	41
10	Dynamic memory ..... Pointer types, new, dispose.	52
11	Sets ..... Declarations, set operations.	59
	Appendix A ..... Information about the Database program.	62

Preface

This section is intended to be a tutorial for Pascal programming. It was specifically designed as a learning aid for TRS-80 Pascal, and is an intermediate level tutorial guide. It is assumed that the reader has had some programming experience. This tutorial is an excellent teaching aid for most other Pascals because TRS-80 Pascal is an implementation of standard Pascal. Any extensions to the language are covered in the TRS-80 Pascal Reference Manual. In this book the standard Pascal referred to is defined by Pascal USER MANUAL AND REPORT(2nd edition) by Kathleen Jensen and Nikalus Wirth (Springer-Verlag, 1975). People with some exposure to BASIC or other programming languages should have no trouble understanding the explanations or example programs. It may be helpful to refer to the TRS-80 Pascal reference manual, for additional details and answers. This tutorial was designed to be as clear and precise as possible for the newcomer to Pascal. It avoids all tricky and confusing explanations, and in many cases includes program segments as examples. This greatly reduces the clutter that often gets in the way of learning computer languages.

The first chapter examines the major advantages of Pascal as a general programming language. You may skip this section and begin reading chapter two if you wish. However, there are many important aspects about Pascal that are explained in chapter one. This tutorial will provide a logical and structured approach to learning . After all, that's what Pascal is all about.

INTRODUCTION

Pascal was created by Professor Nicklaus Wirth at the Swiss Technical Institute in Zurich Switzerland. It was first announced in 1965 when the most popular programming languages in use by the computer industry were Fortran and Cobol. In teaching environments, like Universities, Algol was a popular language for introducing students to computer programming. Wirth felt that languages like Fortran and Cobol were too loosely structured to promote good programming habits to students. Algol, although more structured, had significant drawbacks. Wirth decided to depart from normal teaching practice and designed a new language patterned after Algol, to be his new teaching language.

Pascal inherits the structured control statements of Algol and adds powerful data structuring capability. The language was designed to promote good programming practices and encourage clarity and modularity in programs. Since the first implementation of Pascal on the CDC-6600 computer system in 1971, Pascal has proven to be one of the most popular programming languages in existence.

Pascal has the distinction of being created for the purpose of making the development of computer programs a structured and logical process. Pascal contains the best features of most high level programming languages. Many college instructors at major universities today use Pascal or Pascal like languages to teach structured programming classes. Structured programming classes emphasize the use of guidelines and rules for developing computer programs. Some of the goals of structured programming are to encourage modularity and functionality, promote good documentation and to generate programs that have smooth flows of logic from the beginning to end. Programs are usually developed in Pascal or an English like Pascal and then hand translated to any available computer language such as Basic, for execution.

Although the implementation language may not be highly structured, the final program will be more clear and readable. Indeed, that is exactly what most Pascal programmers do when they need to use other languages. However, this is no replacement for implementing the program in Pascal, as there are no translations for the rich and powerful data structures and many other features that exist in Pascal.

Data types and structures are two important features of the language. They comprise one of the largest differences between languages such as Pascal and BASIC. Most BASIC programmers are familiar with the data types integer and real. A data type is simply the kind of information that may be stored in a variable. Pascal includes nine predefined types: char, integer, real, set, file,array,record, boolean and text plus an infinite variety more, as you may invent data types at will.

Data structure is another name for a variable type such as the array. Pascal allows you to build new data structures as desired. The use of record data structures can be very powerful when building or maintaining data bases. With one simple output statement, an entire data structure may be written to a file.

Variables are assigned storage only as needed during program execution, thus reducing demands on memory. They also may have names with as many characters in them as desired provided that the first 8 characters form a unique name. Long names don't require any more storage space than short ones.

Extra spaces,tabs, and carriage control may be placed freely in a source program, except in the middle of identifiers and character strings. An identifier is defined to be a program, variable, constant, type, procedure or function name. Comments may be inserted anywhere spaces are allowed and are delimited by (* *) or { } . These features don't affect the speed or the size of the final program, and greatly improve readability.

The concept of local variables is important. Variables declared in this manner will have restricted access by other parts of the program. This can prevent accidental changes in their values.

If there are a series of statements that need to be executed by different sections of the program, they may be placed in a procedure or function declaration. A procedure or function is just a collection of program statements that may be called to perform their task at various times during the program. Repetitive programming may be prevented by creating libraries of commonly used procedures or functions. Parameters may be passed to these subroutines by "value or reference".

When a parameter is passed by reference, the actual parameter is passed to the procedure, and if the procedure alters its value, the parameter's value is changed in the rest of the program. When a parameter is passed by reference , the argument must be a variable.

When a parameter is passed by value, what is passed is a copy of the argument. If the procedure alters the parameter's value, the value in the rest of the program is not changed. When a parameter is passed by value, its argument may be a variable or any legal arithmetic expression. Parameters passed by value can prevent accidental changes in a value by procedures.

A careful use of procedures and functions will make the program more readable and will eliminate branching statements that are difficult to follow.

The logical operators AND , OR, and NOT along with the relational operators: greater than ">", less than "<", equal "=", not equal "<>", greater than or equal ">=", less than or equal "<=" are available in Pascal. Statements like : IF(count < 10) and (not FAILURE) then "do the following", make control statements very clear.

There are six statements in Pascal used for the flow of control. Loop control is performed by the FOR, REPEAT and WHILE statements. Conditions are tested with the IF and CASE statements. Branching is accomplished by the GOTO statement.

Program execution speed may be of particular importance in certain applications. TRS-80 Pascal programs execute between 10 and 50 times faster than most interpreted Basics on the same computers. In fact, they are significantly faster than many other Pascal implementations.

As a general programming language, Pascal has the following advantages.

(1) The powerful ability to build new data types and structures as desired.
(2) The control statements while, repeat, for, if, case and goto.
(3) The logical operators AND, OR, NOT.
(4) The relational operators: equal to, less than, greater than, less than or equal to, greater than or equal, not equal to.
(5) Recursive procedures and functions with parameter lists.
(6) The ability to insert blanks and comments in the source program easily,and long variable names, with no space or time penalty.
(7) User controlled dynamic memory management.
(8) Efficient memory management of variables,functions and procedures.
(9) Arrays of one or more dimensions.
(10) Record data structures.
(11) Sets and set operations.
(12) Subrange and enumerated data types.
(13) Named constants.
(14) Read and write statements plus formatted write statements.
(15) Built in functions and procedures.

TRS-80 Pascal has the added advantage of being a full implementation of standard Pascal, thus program portability is greatly enhanced. These features, and the fact that programs generated by TRS-80 Pascal execute much faster than programs generated by most BASIC or other Pascal systems, make TRS-80 Pascal a logical choice as a general high level programming language.

STARTING CONCEPTS

At the simplest level of structure of a Pascal program are the program, begin, and end statements. They may be thought of as the outer shell that must be around all programs. The actual program is placed between these begin and end statements. Example:

Listing 1.1

```
PROGRAM test;
BEGIN
END.
```

This is a completely legal Pascal program although it actually does nothing. We can modify it by adding a writeln statement to it.

Listing 1.2

```
PROGRAM test;
BEGIN
  WRITELN(OUTPUT,'* Pascal is a very structured language.');
  WRITELN(OUTPUT,'* It promotes good programming habits.');
END.
```

The program will write to the file associated with OUTPUT the following message.

```
* Pascal is a very structured language.
* It promotes good programming habits.
```

The two writeln statements comprise the only action in the program. The OUTPUT in the writeln tells the computer to write the message to the file associated with the logical name OUTPUT. How this association is accomplished is a computer dependent process, and is explained in the System Implementation Manual. The string in single quotes is a text string that may be composed of printable characters. Notice two things about this program. First, the text string may not be broken up across line boundaries, however blanks may be used freely elsewhere to make the program more readable. Secondly, a semi-colon is required after each writeln statement. In fact, semi-colons are required after most Pascal program statements. For now, a good rule of thumb is to always include a semicolon after legal Pascal statements. The program name is test, but may be any identifier where the starting character is a letter. The "." must always occur after the last END statement in the program.

Another output statement similar to the writeln statement is the write statement. In the first sample program the two messages were written to different lines on the file. The writeln statement caused the file position pointer to reposition to the beginning of the next line after each message was written. The file position pointer is another name for the cursor when the file I/O is directed to the terminal. The write statement, does not reposition the cursor after the message has been written. Instead, the cursor remains at the end of the last message, and the next text will appear on the same line. The cursor represents the point on a line where text will appear from the next write statement.

Listing 2.1

```
PROGRAM test;
   BEGIN
   (* the purpose of this program is to give an example *)
   (* of how to use the WRITE and WRITELN procedures    *)
   WRITE(OUTPUT,' * Now is the time');
   WRITE(OUTPUT,' for all good programmers');
   WRITE(OUTPUT,' to learn');
   WRITELN(OUTPUT,' Pascal.');
   (* The next statement starts on a new line *)
   WRITELN(OUTPUT,' * You will become a Pascal magician.');
   END.
```

The following message will be written to output.

```
* Now is the time for all good programmers to learn Pascal.
* You will become a Pascal magician.
```

If you noticed, the text enclosed between the (* *) did not affect the program execution. They are simply comments by the programmer to help clarify the logic in the program. Comments may be especially helpful later when you have forgotten how the program functions. They may be inserted anywhere except in the middle of identifiers or text strings. An identifier is just another name for a program, variable, constant, procedure or function name. Procedures and functions will be explained later.

Tutorial Quiz 2.0

(1) The first statement of a Pascal program must be the _____ statement.

(2) The _____ statement will not move the cursor to beginning of the next line.

(3) The _____ statement will move the cursor to the beginning of the next line.

(4) Most Pascal statements are followed by a _____.

(5) The ____ statement must be the last statement of a program.

(6) Quoted _____ may not be broken up across line boundaries.

Answers:

(1) program (2) write (3) writeln (4) semicolon

(5) end (6) strings

DATA CONCEPTS

Variables

Variables in Pascal serve the same purpose as they do in most other programming languages. They serve as storage areas for the information that the programmer may wish to manipulate. These storage areas are referred to by names that are chosen by the programmer. Each variable name must start with a letter. It may be composed of any combination of letters and digits, although in many Pascal implementations, the first eight characters must form a unique name within the program.

Reserved words

There are certain words in Pascal that have special meanings. These words are called reserved words, and variables may not have these names. For a complete list see the TRS-80 Pascal Language Reference Manual.

Variable types

Variables must have associated with them a specific type. The type is the kind of information that is going to be stored in that variable. For example, the variable "taxnumber" may represent a business tax number. This taxnumber might take on the numerical value of 1 to 100 at any time in the program. This would be an example of the type, integer.

Declaring variables

All variables must have their specific type declared in a special section of Pascal programs called the var section. There are five predefined variable types in Pascal that we will concern ourselves with at this time. They are integer, real, char, text and boolean. The var section of a program consists of the word VAR followed by any number of variable declarations. A variable declaration has the form of variable name: variable type; . A colon separates the variable name from the variable type, and a semicolon must follow each variable declaration.

Integer variables

The type integer may be used to represent whole numbers. The minimum and maximum size allowed by Pascal is computer dependent, but on many micro computers they range from -32768 to +32767 . The following is a program example of a variable declared as an integer. Notice that a colon is required to separate the variable name taxnumber, from the variable type, integer.

Listing 3.1

```
PROGRAM test;
VAR
taxnumber:INTEGER;
BEGIN
END.
```

Real variables

The type real may be used where a variable must store numbers that may have fractional or decimal values. The numbers 2.98 , 3.047 , 0.0009 , 0.009 and 37.0998 are all examples of real numbers. Real numbers must start with a digit and may contain a decimal point. If a decimal point is present, a digit must follow the decimal point. The numbers .009, 10. are illegal real numbers, as there is no digit before and after the decimal point. The size and precision of real numbers are computer dependent . Real variables may represent the dollar selling price of some product by a store, or an entry into your checkbook. They are declared as follows:

Listing 3.2

```
PROGRAM test;
VAR
  taxnumbr:INTEGER;
  cost    :REAL;
BEGIN
END.
```

Note that the indentation of the declaration section does not affect the execution of the program.

Char variables

If a variable is declared as a char type, then it may represent a single character such as the character 'A'. In Pascal, the characters may be composed of letters, digits and other special symbols. If a digit is to be referred to as a character instead of a number, it is enclosed in single quotes like the character string was in program listing 2.1. The only difference is that a char variable may only represent one character at a time.

Text variables

Variables declared to be of the type text are used to direct output or input information to files on disks, or to other devices. Text is predefined to be a special file of char.

Boolean variables

A variable declared as the type boolean may only have two values. They are true and false. This kind of variable is primarily used in flow control statements. Boolean variables are typically used in the WHILE, IF or REPEAT control statements. These statements will be covered in later chapters.

Const section

Often, specific variables will have fixed values during program execution. In this case you may declare these values as constants. In Pascal, they are declared in the CONST section. The const declaration section is placed between the program and the first begin statement of the program. Constants may have names like variables do. In fact their names should reflect their nature. Constants may be integers, real numbers or a text string. A text string constant is any character string enclosed between single quotes. A string constant generally may be used anywhere a packed array[1..n]of char variable may be used. This variable type will be explained later.

(1) _____ serve as storage areas for information that the programmer may wish to manipulate.

(2) Variable types are declared in the ___ section of the program.

(3) Five predefined type of variables in Pascal are _____ , _____ , _____ , _____ , _____ .

(4) The syntax of a variable declaration is :
var variable name : _____ ;

(5) Variables declared as the type _____ may take on the value of letters, digits and other special symbols.

(6) A variable declared to be of the type _____ is used to direct I/O to files.

(7) A value that is fixed in the program and will not change may be declared as a constant in the _____ section of the program.

Answers:

(1) Variables (2) type (3) char, integer, boolean, real, text

(4) type (5) char (6) text (7) const

ADVANCED I/O

Procedures rewrite, writeln.

Can you guess what this program will do if you run it ?

Listing 4.1

```
PROGRAM alpha;
CONST
  pi      = 3.141597;
  maxtax  = 2000;
  tstring = ' I am a  Pascal Wizard';
VAR
  out   :TEXT;
  max   :REAL;
  number:INTEGER;
BEGIN
   REWRITE(out);
   WRITELN(OUTPUT,'Program starting execution.');
   WRITELN(' The value pi = ',pi);
   WRITELN(' The value maxtax = ',maxtax);
   WRITELN(tstring);
   WRITELN(out,'This program tests file I/O');
   WRITELN(OUTPUT,'Program finished.');
END.
```

From example 2.1 you already know that the first and last writeln statement will cause the program to direct the messages to the file associated with output. The following message will be written to output.

```
Program starting execution.
 The value pi = 3.14159
 The value maxtax =       2000
 I am a Pascal wizard
Program finished.
```

The message, "This program tests file I/O", will be written to the file associated with out.

Examine the first writeln statement. In the specific case where the first argument for the writeln statement is output, the user is not required to declare output in the var section as with other files. Notice also that there is no output argument in the second,third and fourth writeln statements. In Pascal, it is not required to have output as an argument. Output is a default argument. Ie; the statements writeln(output,' help'); and writeln(' help'); are equivalent in Pascal. In Pascal the write and writeln statements may have multiple arguments. The first argument always directs the I/O operation to a specific file except for the case previously explained. In listing 2.1 the two arguments were output and a text string. Constants and variables may also be arguments. The values of the variables and constants will be written in the same order as they appear in the argument list.

Rewrite statement

The purpose of the rewrite(logical filename) statement is to open a file on some hardware device, and ready it for writing. Note that the previous contents of any file used in a rewrite statement will be lost. The specifics of how to associate the logical filename in parentheses with a physical filename is implementation dependent and is explained in the TRS-80 Pascal System Implementation Manual. Standard Pascal does not require the file output to have a rewrite performed on it before it is written to. Output is the only file in Pascal that does not require a rewrite before it is written to. It is predeclared to be a textfile by Pascal.

Reset statement

The purpose of the reset statement is to ready a file for reading to a program. A reset (logical filename) statement will open the physical file associated with the logical filename and read the first line. In TRS-80 Pascal, the first line is not read until required by an EOF or EOLN function call. These functions will be explained later. All files that are to be used for reading must be reset, except Input. Input is a predeclared textfile within Pascal.

Read, readln statements

The read statement is similar to the write statement, except that its purpose is to read information into the program instead of to write information. The read statement will read a value into a variable from a file and will leave the cursor at the last character read.

Specific reads on the same file will cause a series of inputs to occur from the same line. When a read is performed on an integer or real quantity in a text file, the read will start scanning the line until any non-blank character is found. The next contiguous non-blank characters will be interpreted by the read as the input value. If another read is performed on the same file, the read procedure will scan forward and repeat the process, until the end of line is reached. If the end of line is reached before any integer is found, the scan will continue at the beginning of the next line.

The readln statement performs the same function as the read statement, except that the cursor will always be positioned to the beginning of the next line after all inputs to the read statement are satisfied, even if the end of line has not been reached. The readln statement is not required to have arguments. The effect of such a readln is to position the cursor to the beginning of the next line without reading any values. The arguments allowed for the read statement are variables. As with OUTPUT in the write statement, INPUT is predeclared to be a text file. If a read statement does not have a file argument, it is assumed to be the predeclared file INPUT.

Try running the following program. It will give you a little more experience performing program I/O.

Listing 4.2

```
PROGRAM testIO;
(* Purpose- the purpose of this program is to          *)
(* demonstrate I/O to a text file using integer and    *)
(* real input variables.                               *)
VAR
   taxnumbr,emnumber    :INTEGER;
   tax                  :REAL;
   ID                   :PACKED ARRAY[1..72]OF CHAR;
BEGIN
   WRITELN(OUTPUT,'* Enter your federal tax number: ');
   READLN(INPUT,taxnumbr);
   WRITELN(OUTPUT,'* Enter your dollar tax total: ');
   READLN(INPUT,tax);
   WRITELN
      (OUTPUT,' * Enter your employee number,a space,');
   WRITELN(OUTPUT,' followed by your business ID number:');
   READ(INPUT,emnumber);
   READLN(INPUT,ID);
   WRITELN(OUTPUT,' Tax number       = ',taxnumbr);
   WRITELN(OUTPUT,' Dollar tax total = ',tax);
   WRITELN(OUTPUT,' Employee number  = ',emnumber);
   WRITELN(OUTPUT,' Business I.D.    = ',ID);
END.
```

The following I/O will occur at the terminal if the filename associated with input and output is the local terminal.

```
* Enter your federal tax number:
32000                          <user input>
* Enter your tax total:
2345.98                        <user input>
* Enter your employee number,a space,
followed by your business ID number:
23455 4669                     <user input>
Tax number          = 32000
Dollar tax total    = 2345.98
Employee number     = 23455
Business I.D.       = 4669
```

Tutorial Quiz 4.0

(1) The predefined file variables _____ and _____ are not required to be declared in the var section as the type text.

(2) The first argument in a _____ , _____, _____, _____ statement directs I/O to a file or device.

(3) The purpose of the _____ statement is to open a file and ready it for writing.

(4) The purpose of the _____ statement is to open a file and ready it for reading.

(5) After a _____ , the previous contents of the file are lost.

(6) A _____ or _____ statement will cause the cursor to move to the next line after execution.

Answers:

(1) input, output (2) read, readln, write, writeln

(3) rewrite (4) reset (5) rewrite (6) readln, writeln

STATEMENTS

Assignment statements

From previous examples, you know how to read a value into a variable and how to write it. Now you will learn how to alter its value within the program. The statement that does this is the assignment statement. It allows you to set a variable's value equal to an expression. An expression may be a variable name or a series of arithmetic or boolean operations. A simple assignment statement takes the form of variablenamel := variablename2; . The " := " operator causes the variable on the left hand side to become equal to the value of the variable on the right hand side.

Listing 5.1

```
Program MAGIC;
VAR
   intrate,principle,anint,calc:REAL;
BEGIN
   WRITELN(' ******* Interest rate problem *******');
   WRITELN(' Enter annual interest rate:');
   READLN(intrate);
   WRITELN(' Enter the principle amount of loan:');
   READLN(principle);
   calc:= intrate * principle;
   anint:=calc;
   WRITELN(' Your annual interest payment = ',anint);
END.
```

Arithmetic operators

In the program listing 5.1 you may have noticed the statement "calc:= intrate * principle" . The " * " is the multiply operator in Pascal. There are seven arithmetic operators in Pascal with precedence as follows :

OPERATOR PRECEDENCE TABLE 5.1

Symbol	Precedence	Description
-	(1) Highest	Unary operator. Negates a single argument.
*	(2)	Multiplies two arguments
/	(2)	Divides two real arguments
div	(2)	Divides two integer arguments
mod	(2)	Divides two integer arguments and keeps the remainder as the result.
+	(3) Lowest	Adds two arguments
-	(3) Lowest	Subtracts two arguments

Operator precedence

If an arithmetic expression is composed using different operators without any parentheses, the order of evaluation is based on the above table, where operations with the highest precedence are performed first. Any operations at the same level are performed in left to right order.

Parentheses

In Pascal, this natural order of precedence may be altered by enclosing a portion of the expression in parentheses. The parentheses has the highest precedence of all operators. Parentheses may be nested to alter the evaluation sequence as desired. In this case, operations buried deepest within are evaluated first.

The following program will illustrate the use of the arithmetic operators and parentheses.

Listing 5.2

```
PROGRAM math;
CONST
   fudge    = 100;
   lossacre = 0.50;
VAR
   acsoy,acgreen   :INTEGER;
   prsoy,prgreen   :REAL;
   profit,overcost :REAL;
BEGIN
   WRITELN(OUTPUT,' **** Farmers profit analysis program **** ');
   WRITELN(OUTPUT,'* Please enter the following information:');
   WRITELN(OUTPUT,'* Acres planted in soy beans = ');
   READLN(INPUT,acsoy);
   WRITELN(OUTPUT,'* Profit per acre of soybeans = ');
   READLN(INPUT,prsoy);
   WRITELN(OUTPUT,'* Acres planted in green beans = ');
   READLN(INPUT,acgreen);
   WRITELN(OUTPUT,'* Profit per acre of green beans = ');
   READLN(INPUT,prgreen);
   WRITELN(OUTPUT,'$$$ COMPUTATION IN PROGRESS $$$');
   profit:= acsoy * prsoy + acgreen * prgreen
            - (fudge / (acsoy+acgreen) * lossacre);
   WRITELN(OUTPUT, ' Your computed profit is ');
   WRITELN(OUTPUT,profit);
END.
```

The profit calculation uses parentheses to alter the normal operator precedence. If the normal precedence is followed, the calculation will yield the wrong result.

The order of evaluation without parentheses would be:

```
(1) acsoy and prsoy multiplied.
(2) acgreen and prgreen multiplied.
(3) fudge / acsoy
(4) acgreen * lossacre
(5) result1 + to result2
(6) result5 - result3
(7) result4 + result 6
```

The desired result is obtained by including the parentheses as in the example. The apparent order of evaluation would be:

```
profit := acsoy * prsoy + acgreen * prgreen
       -(fudge / (acsoy+acgreen) * lossacre) ;
```

```
(1) acsoy added to acgreen.
(2) fudge / result1
(3) result2 * lossacre
(4) acsoy * prsoy
(5) acgreen * prgreen
(6) result4 + result5
(7) result6 - result3
```

If two numbers are operated on, the normal result will have a type that is dependent on the argument types. The variable types required to store the results of specific operations are summarized in the following table.

*	multiply	real * integer = real result. integer * real = real result. real * real = real result. integer * integer = integer result.
/	real divide	real / real = real result. real/integer = real result. integer/real = real result. integer/integer = real result

div	integer divide	integer div integer = integer result. integer arguments only.
mod		integer mod integer = integer (integer div integer= remainder)
+	add	integer + integer = integer result. integer + real = real result. real + integer = real result. real + real = real result.
-	subtract	integer - integer = integer result. integer - real = real result. real - integer = real result. real - real = real result.

Compound statements

If a series of program statements are surrounded by a begin and end statement, then the enclosed statements are considered a compound statement. Compound statements are normally used as arguments to control structures such as the WHILE and IF. A compound statement may occur by itself anywhere in a Pascal program, however, its meaning would be the same as if the begin and end were not present. The important thing to remember about Pascal is that anywhere a single statement may be used, a compound statement may be used.

(1) ____ is the symbol for the assignment operator.

(2) If a series of statements are surrounded by a begin and end , it is called a _____ statement.

(3) Operator precedence refers to the order in which an _____ is evaluated.

(4) The natural order of expression evaluation may be altered by using _____ .

(5) The _____ with the highest precedence will be evaluated first.

(6) Operators that have the same level of precedence will be evaluated in _____ to _____ order.

(7) After executing the following Pascal statement, variable x will have the value _____.

```
PROGRAM QUIZ;
VAR
   x:integer
BEGIN
   x:=4 + 5 * 2;
END.
```

Answers:

(1) := (2) compound (3) expression (4) parentheses

(5) operator (6) left, right (7) 14

FLOW CONTROL

FOR statements

If you wish to execute a series of statements a predetermined number of times, you should use the FOR statement. The for statement will cause a single or compound statement to execute a specific number of times. Examine the following example .

Listing 6.1

```
PROGRAM math;
CONST
   fudge    = 100;
   lossacre = 0.50;
   prsoy    = 195.98;
   prgreen  = 200.56;
VAR
   acsoy,acgreen,nofields,select,fieldnumber:INTEGER;
   profit,overcost: REAL;
BEGIN
   WRITELN(OUTPUT,'* Farmers planting analysis program * ');
   WRITELN(OUTPUT,'* How many fields do you have ?');
   READLN(INPUT,nofields);
   FOR fieldnumber := 1 to nofields DO
      BEGIN
      WRITELN(OUTPUT,'* For field number ',fieldnumber);
      WRITELN(OUTPUT,'* Acres planted in soy beans = ');
      READLN(INPUT,acsoy);
      WRITELN(OUTPUT,'Acres planted in green beans = ');
      READLN(INPUT,acgreen);
      profit:= acsoy * prsoy + acgreen * prgreen
               - (fudge / (acsoy+acgreen) * lossacre);
      WRITE(OUTPUT, '* Your computed profit for field number  '
                   ,fieldnumber,'   is ');
      WRITELN(OUTPUT,profit);
      END;
END.
```

The loop control variable is "fieldnumber" . This variable is declared as an integer. When the loop starts its execution, "fieldnumber" takes on the value of one for the first pass through the loop. Successive loop iterations cause this value to be incremented by one until its value is greater than "nofields" . At this point, the loop will stop and control will be passed to the next statement in the program. The lower and upper bounds on the loop control variable do not have to be variables or constants, but may be arithmetic expressions. The expression is evaluated one time, at the beginning of the loop. The upper bound must be greater than or equal to the lower bound for the loop to execute at least once.

A variation on the for loop just described causes the loop control variable to be decremented by one instead of incremented by one. The syntax for this is the same as above except that the "to" in the for statement is replaced with "downto" . The initial upper bound on the loop control variable must be larger than or equal to the lower bound for the loop to execute at least once.

Case statement

The case statement is used as a selection control statement. It is used when you need to execute one statement from a list of statements. Notice the following program. In front of every statement in the list, is a case selector constant. This selector value must be of the same type as the case selector variable, and may be composed of a list of values for each statement it precedes. The "end" must follow the last statement in the list in order to terminate a case statement. We will be concerned with selector variable of type integer at this time.

Listing 6.2

```
PROGRAM moonphase;
CONST
   dayphcorr  = 10;
   lencycle   = 28.3;
VAR
   daynumber,intphase                :INTEGER;
   startphase,phase,month,day,year :INTEGER;
   realphase,phasecorrection         :REAL;
BEGIN
   WRITE(OUTPUT,' *** Lunar Phase calculation program');
   WRITELN(OUTPUT,' ***');
   WRITELN(OUTPUT,' Enter the month/day/year:');
   READLN(INPUT,month,day,year);
   startphase := ((year-78) * 365) + dayphcorr ;
   CASE month of
     1: daynumber:=1;
     2: daynumber:=32;
     3: daynumber:=60;
     4: daynumber:=91;
     5: daynumber:=121;
     6: daynumber:=152;
     7: daynumber:=182;
     8: daynumber:=213;
     9: daynumber:=243;
    10: daynumber:=274;
    11: daynumber:=304;
    12: daynumber:=334;
   END;    (*case*)
   startphase := startphase + daynumber + day;
   realphase := startphase / lencycle;
   intphase := TRUNC(realphase);
   realphase:=realphase-intphase;
   phase:=realphase * lencycle;
   CASE phase OF
   1,2,3,4,5,6,7         : WRITELN(OUTPUT,
                            'The moon is in its first quarter.');
   8,9,10,11,12,13,14    : WRITELN(OUTPUT,
                           'The moon is in its second quarter.');
   15,16,17,18,19,20,21 : WRITELN(OUTPUT,
                           'The moon is in its third quarter.');
   22,23,24,25,26,27,28 : WRITELN(OUTPUT,
                           'The moon is in its fourth quarter.');
   END;        (*case*)
END.           (*PROGRAM*)
```

The purpose of the program in listing 6.2 is to compute the phase of the moon. Several examples of case statements are used with differing case selector lists. The calculations are based on a known starting phase of the moon at some past day,and year. The initial startphase calculation yields the number of days since this known starting date as a function of the number of years, corrected for the starting phase of the moon. The remainder of the calculations simply adjust this value to yield the whole number of days since the known starting phase, then divide the resultant number of days by the lunar cycle length in days. This program does not consider the effect of leap years. Notice that mixed mode expressions consisting of real and integer arithmetic are used throughout the calculations. A careful study of the previous type result tables will verify their validity. Notice that the value of realphase is used as an argument for the TRUNC function. This is a predefined function available in Pascal that will truncate a real number and store the result in an integer.

Tutorial Quiz 6.0

(1) The ___ statement is used to make a single or compound statement execute a specific number of times.

(2) In successive loop iterations in a _____ loop, the loop control variable is either incremented by one or decremented by one.

(3) The _____ statement is used to select a statement to execute from a list of statements.

(4) The "downto" and "to" are elements of the _____ statement.

(5) An ___ must follow the case statement.

Answers:

(1) for (2) for (3) case (4) for (5) end

DECISION TESTING

Often, it is necessary to make tests to determine the flow of control in a program. The case statement is a simple example. However, it may become necessary to perform more complex tests than the case statement was intended for. Pascal has a powerful set of logical and relational operators that make such testing easy. Most logically complex programs use relational testing for advanced control. The logical and relational operators are as follows :

Logical operators

and - Will evaluate two boolean expressions, then perform a logical "and" on them, returning either a boolean "true" or "false".

or - Will evaluate two boolean expressions, then perform a logical "or" on them, returning either a boolean "true" or "false".

not - Will change a boolean value to the opposite value.

Relational operators

It is often necessary to compare several variables for equality in an expression to determine the flow of control. This may be accomplished by relational testing. There are six relational operators in Pascal, all with equal precedence. Their precedence may be altered just like the arithmetic operators by the use of parentheses. If the relational test fails, a Boolean False is returned by the expression. If the test succeeds, then a true is returned. The operators are as follows.

=	Equal to	<>	Not equal to
>	Greater than	<	Less than
>=	Greater than or equal to	<=	Less than or equal to

There are two constructs in Pascal that often use relational testing for loop control. They are the while and repeat statements. Almost all goto and other branching constructs may be replaced with these statements. Unlike the goto statement, these statements force simple and clear design of loops, often eliminating the unclear conditions for exiting. Usually, if it is not possible to formulate a loop construct using the while, repeat and if statements, instead of a goto, it is because the loop itself has not been properly defined. Ie; the programmer does not have the specifics clear in his mind.

If statement

A typical use of a relational test is illustrated in the if statement. In the following example let the variables "Monday" and "October" be of the type boolean with their values both being true.

Listing 7.1

```
PROGRAM testIF;
VAR
 Monday,October:BOOLEAN;
BEGIN
   Monday:=true;
   October:=true;
   IF October AND Monday THEN
      WRITELN(OUTPUT,'Its October and Monday')
   (*notice no semicolon after the previous statement*)
   ELSE WRITELN(OUTPUT,'Date unknown.');
END.
```

This program will print the message, "Its October and Monday" since October is true, and Monday is true. This example illustrates the use of the " if then else " statement in Pascal. If the expression is evaluated to be true, the first action will be taken. If it is false, the statement following the else will execute. The statements may be simple or compound. Notice that a semicolon may not precede the else in the IF statement.

Notice the following example where "income" has been declared as the type integer and "president" is of type boolean.

```
IF (income > 32000) AND NOT(president) THEN
   BEGIN
   WRITELN(OUTPUT,'You are being audited by the IRS.');
   WRITELN(OUTPUT,'Please justify your deductions.');
   END;
```

The value of the expression will be true if the integer value of "income" is greater than 32000 and the boolean value of "president" is false. When the value of "president" is false, the not operator will reverse its value to true. This type of expression is one of the strengths of Pascal. With a little experience, you will find it easy to write expressions. This greatly improves the readability of logically complex programs. Arguments for relational operators must be of the same type. In the example, "income" must be declared as an integer type for the statement to be valid in Pascal. For now, we will concern ourselves with integer and boolean comparisons.

While statement

The while forces a statement to execute while some condition is satisfied. The condition is the value of a boolean variable or the boolean result of some expression. Some computation inside the loop should change one of the variables used in the test to cause the relational test to fail, terminating the loop. The while statement will perform the test at the beginning of every loop. The while loop might never execute any of the enclosed statements as the initial test occurs before the loop is entered. In the next example, cnt, cost and unitprice are declared as type integer, and underbudget is of type boolean. Notice the following example syntax.

```
cnt:=0;
underbudget:=true;
WHILE (cnt < 20) AND (underbudget) DO
        BEGIN
        cnt:= cnt + 1;
        cost := cnt * unitprice ;
        IF (cost > 200 ) THEN underbudget :=false;
        END;
```

The previous example will execute as a conditional loop instead of a predetermined number of times as in the for loop. When "cnt" gets incremented to twenty one, or cost exceeds 200, the loop will terminate. Note that the "cost > 200" test could have been put in the while expression just as easily.

Repeat statement

Another statement similar to the while is the repeat. A statement or series of statements will be repeated until an expression becomes true. The difference between the while and repeat may not be obvious. The difference is that the repeat statement will always execute at least once because the relational test occurs at the end of the loop. The use of repeat sometimes causes problems for new programmers, as there may be cases where you do not want the loop to execute at all, however it will always execute at least once. An example of repeat is as follows:

```
cnt:=0;
underbudget:=true;
REPEAT
   cnt:=cnt + 1;
   cost := cnt * unitprice;
   inventory:=inventory +1;
   IF (cost > 200) then underbudget:=false;
UNTIL(cnt >= 20) OR NOT(underbudget);
```

Notice that the test was changed to use the OR operator instead of the and operator. This is simply due to the different context of the two statements. There is no begin or end required. The statement(s) to be executed are simply placed between the repeat and until. What happens to this loop if the initial value of "unitprice" is greater than 200 ? The loop will terminate on the first iteration, but alters the value of "inventory". This might not be the desired result and could cause an illegal entry into the inventory. In this situation, the while statement would be the proper choice of a looping construct, as it would detect this before "inventory" is changed.

Tutorial Quiz 7.0

(1) The logical operators in Pascal are: and, ___ , ___ .

(2) The and operator will return a value of ____ , if the value of the both expressions it is evaluating is true.

(3) The or operator will return a value of _____, if one of the expressions it is evaluating is true.

(4) The not operator will reverse the value of a _____ variable or expression.

(5) The IF statement will execute the else portion of the statement, if the value of the expression is _____.

(6) The while statement will execute as long as the boolean result of the expression is _____.

(7) The repeat will execute all statements between the repeat and until as long as the expression is _____ .

Answers:

(1) or, not (2) true (3) true (4) boolean (5) false

(6) true (7) false

Chapter 8

PROCEDURES AND FUNCTIONS

Procedures

In the introduction, one of the claimed strengths of Pascal was that it promotes modularity. Modularity is another name for organizing a program into sections, each of which performs a specific function, instead of one large block of continuous statements. One of the reasons that Pascal programs may have a high degree of modularity is that the language was designed with procedures and functions in mind. In a few languages, they are not even supported, and in others, passing parameters can become a major chore. This is not the case in Pascal, as several different methods are available to pass data to subroutines that need it. Furthermore, there are rules about how procedures may call other procedures and access their internally defined variables. These scoping rules, as they are called, may seem a little restrictive, but they provide valuable protection. This partitioning of the problem eventually decreases program size and improves readabilty to the programmer or anyone who must maintain it. A simple way to decide whether a procedure or function should be used is to examine the problem and to decide if there are a series of statements that need to be executed several times, and in different parts of the program. The identified program segments should be placed in a procedure or function.

Procedure structure

Procedures may be thought of as complete sub-programs that have data passed to them as needed. In many descriptions written about Pascal, they are often called one of the basic blocks, and in this manual, a block will be considered to be a program, procedure or function. The structure of a procedure is the same as for the original program with a few exceptions. The data that is passed to a procedure block is passed through a parameter list. The parameter list is placed after the procedure name. Examine the following program.

```
Listing 8.1
PROGRAM INSTRUCTIONAL;
VAR
   number   :INTEGER;
   posnumber:INTEGER;
   legal    :BOOLEAN;
PROCEDURE readn ( VAR number: INTEGER; VAR legal: BOOLEAN );
(* The purpose of this routine is to read       *)
(* a positive number from a file in a           *)
(* character format and convert it to an integer*)
(* format.                                      *)
VAR
    loopcontrol,forcntr,inc:INTEGER;
    string                 :ARRAY [1..72]OF CHAR;
BEGIN
    FOR loopcontrol :=1 to 72 DO string[loopcontrol]:=' ';
    loopcontrol :=0;
    WHILE NOT EOLN(INPUT) DO
       BEGIN
       loopcontrol := loopcontrol + 1;
       READ(string[loopcontrol]);
       IF(string[loopcontrol]=' ')THEN
         (* Remove all leading blanks from array *)
         loopcontrol:=loopcontrol - 1;
       END;
    number:=0;
    inc:=1;
    FOR forcntr :=loopcontrol  DOWNTO 1 DO
       BEGIN
       number:=number+((ord(string[forcntr])-ord('0'))*inc);
       inc:=inc*10
       END;
    IF (number < 0) THEN
       BEGIN
       legal := false;
       WRITELN('* Error - Illegal entry. Try again. ');
       END
   ELSE legal:= true;
END;              (*procedure readn*)

BEGIN
legal:=false;
WHILE NOT legal DO
   BEGIN
     WRITELN('Enter any positive number:');
   READN(posnumber,legal);
   END;
END.
```

The purpose of the program 8.1 is to read a positive integer in from the file input and to check for illegal entries. This declared procedure represents a typical use for a procedure, since it might be called several times, from different places in the program. Notice the eoln(input) . Eoln is a boolean function that will return a true value when an "end of a line" of the file specified in the parentheses is reached. As soon as the cursor is moved from this position by another readln, it's value becomes false again. Notice also the function call to ORD. ORD is a Pascal function that returns the internal integer representation of a character.

Since there is only one copy of this procedure in memory no matter how many calls there are, a considerable amount of memory space can be saved. In fact, a procedure's variables do not occupy storage space until the procedure is actually called.

The procedure declaration comes after the const and var section, and before the first begin statement of the block in which it resides. Remember, a block may be a main program, procedure or function.

Local variables

Local variables are declared in a particular procedure, function or program. For example, the variable "forcntr" declared in procedure readn, is local to "readn" and is accessible from "readn" only. However, notice the variable "number" declared in the main program block. Inside the procedure "readn" , the variable "number" may be used without declaring it, since it appears in the calling program. This means that if "readn" is called by the main program and "readn" alters "number", then upon return to the main program, "number" will have the altered value. This side effect can be avoided by declaring "number" again in the procedure block. Then all references to "number" will refer to a different variable. The use of global variables should always be kept to a minimum, so as to minimize any accidental changes in their values.

Procedure parameters

An alternate method of changing global variables within a procedure is to pass them as parameters in a parameter list. This allows different variables to be passed at different times and makes the use of the global variable more visible in the program. The parameter list is placed in the procedure declaration after the keyword procedure. In the parameter list, a variable may be passed by two different methods. These two methods are referred to as passing by reference, or passing by value.

When a parameter is passed by reference, the actual argument is passed to the procedure, and if the procedure alters its value, the argument's value is changed in the rest of the program. When a parameter is passed by reference, the argument must be a variable.

When a parameter is passed by value, what is passed is a copy of the argument. If the procedure alters the parameter's value, the value in the rest of the program is not changed. When a parameter is passed by value, the argument may be a variable or any legal arithmetic expression. Parameters passed by value will prevent unwanted changes in a variable value by the called procedure. Notice the following example parameter list.

```
PROCEDURE test ( date: INTEGER; VAR profit:REAL; cost:REAL);
```

The variables "cost" and "date" will be passed by value. The variable "profit" will be passed by reference. Every time a variable is to be passed by reference, the keyword "var" must precede it, otherwise it will automatically be passed by value.

It is sometimes hard for new programmers to understand the difference between letting variables be global when accessing them in a procedure, versus passing them by reference. There is a major difference, in that different variables may be passed to a procedure. The only stipulation is that the variables must match the parameter list. If they are declared as globals and alterd by a procedure, then all values to be passed to the procedure must be transferred to these global variables. A second major difference is that in large programs, it is often difficult to determine what routines are changing specific variables. Sometimes accidental changes may occur in global variables. These changes are often referred to as side effects.

By adhering to the convention of passing the variables to a procedure, it is easier to determine how procedures alter external variables and to minimize unwanted side effects. Certainly, global variables do have use in Pascal programs, but many new Pascal programmers have a tendency to over use them.

Calling procedures

Procedures are called simply by referencing their name followed by an argument list enclosed in parentheses. The list should be composed of variables of the same type and order as declared in the procedure declaration section.

Functions

Another block in Pascal similar to the procedure is the function. Its internal structure is the same as the procedure with const, var and type sections optional. The purpose of a function is similar to a procedure. A procedure may stand alone as a statement, as the call to "readn" illustrates in program 8.1 . A function may not stand alone. It must be used in an expression, and may be used anywhere a variable can be used. Consider the following program.

Listing 8.4

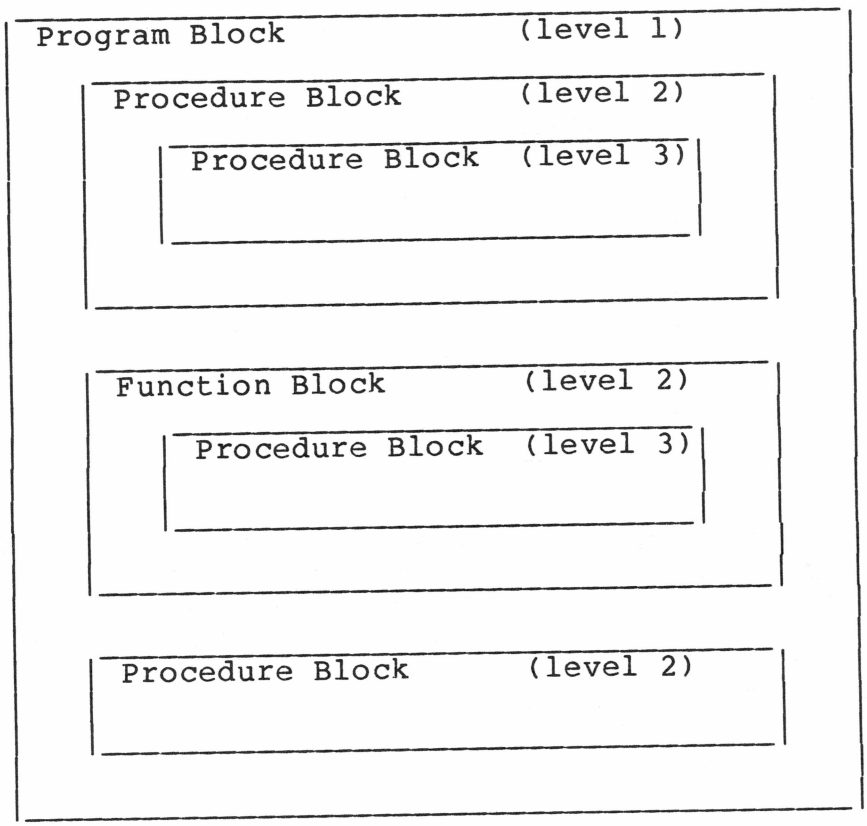
```
PROGRAM functiontest;
VAR
   num:INTEGER;

FUNCTION ABS( number: INTEGER ) : INTEGER;
BEGIN
IF (number < 0) THEN ABS := - number
   ELSE ABS := number;
END;

BEGIN
   num:= -30;
   num := ABS (num);
   WRITELN(' the absolute value of num = ',num);
END.
```

The result returned by the abs function is of the type integer. The result must be used in an expression or assignment statement. It is not valid to simply say abs (num) . The mechanism used to transfer the functions calculated value back to the calling program is the use of an assignment statement to assign the value to an identifier that has the same name as the function name. This particular function is already predefined in Pascal, and serves the same purpose as the example.

Advanced program structure

Pascal is a block structured language. This means that a program is constructed in a block like manner. At a minimum, a program consists of one block. More blocks are created through the use of procedures and/or functions by placing them inside this outermost "program block". The term for this process is called nesting. The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. This provides an appropriate tool for describing scope rules which are discussed later. The block structured organization of a program can be represented pictorially by the following diagram.

Diagram 8.1



A program then consists of at least one block, the program block, and optionally it contains procedure and/or function blocks which are nested within.

Local variables are those variables declared within the var section of a particular procedure. Locals can be accessed from the body of the procedure in which they are declared and from those procedures declared within it. If a variable is used within a procedure and is not declared local to it, then a global variable is used. Global variables are those variables declared in an outer enclosing block.

Listing 8.2

```
PROGRAM globals;
VAR
   i : INTEGER;
   b : BOOLEAN;

PROCEDURE inner;
VAR
  b : INTEGER;
BEGIN
   b := i + 25;
   i := i + 1;
END;

BEGIN
   i := 0;
   writeln(i);
END.
```

In the above program, the i in the procedure refers to the variable i in the main program. Since the program "global" is an enclosing block to the procedure "inner", the variables declared within the program are accessible to the procedure. In the case of the variable "b", the var section of the procedure redeclares b to be an integer. When b is referred to in the procedure inner, the local variable is used. The declaration of b as a local variable "masks" the global definition of b.

Scope rules

The rules of accessability of variables, types and constants are referred to as scope. The scope of an identifier is the procedure in which it is declared, and all procedures declared within that procedure. All identifiers including types, constants, variables and procedure declarations have scope.

If an identifier is redeclared within its scope, the outer definition becomes inaccessible within the scope of the inner definition. In the example above, the declaration of b as an integer within the inner procedure causes all references to b in that procedure to refer to the local variable. The outer definition of b as a boolean cannot be seen.

Pascal requires that all identifiers be declared before they are used. If the declaration of an identifier has not yet been encountered in the text of a program, then the identifier is considered undefined. A procedure can be called from the body of the block declaring it, from the procedures declared within it and from the procedures declared within the same block. However, if procedure A is declared before procedure B in the same block, then procedure B can call A, but procedure A cannot call B. This is due to the fact that the declaration of B has not been encountered in the source text when the body of procedure A is being compiled.

The above visibility restriction can be avoided with the use of forward declarations. In a forward declaration, the body of the procedure is replaced with the word FORWARD. The actual body is then supplied later. If all procedures within a block are declared forward, then any one of them can call any other.

Listing 8.3

```
PROGRAM Outer;
VAR
  i : INTEGER;
FUNCTION Distance(x1, x2 : INTEGER):INTEGER;FORWARD;
FUNCTION Abs(tvalue : INTEGER) : INTEGER; FORWARD;

FUNCTION Distance(*(x1, x2 : INTEGER) : INTEGER *);
BEGIN
   distance := abs(x2 - x1);
END;

FUNCTION Abs(*(tvalue : INTEGER) : INTEGER *);
BEGIN
   if tvalue < 0 then abs := -tvalue
      else abs := tvalue;
END;

BEGIN
   WRITE('DISTANCE = ',Distance(8,2));
END.
```

If a procedure is declared forward, its parameter list is supplied by the forward declaration. The body appears later in the text. The body is introduced by the procedure name followed by a semicolon. The parameter list is not repeated. Notice that in the example, the parameter list is commented out by putting (* *) around it. It is good practice to include the parameter list of a forward procedure in a comment. This makes the body of the procedure easier to read.

Tutorial Quiz 8.0

(1) _____ and _____ promote modularity and functionality in programs.

(2) Data is passed to procedures and functions through a _____ list.

(3) Blocks may be _____ within other blocks.

(4) Nesting affects the _____ of blocks.

(5) A block nested within an outer block may access the outer blocks _____ .

(6) Parameters may be passed by value or _____ .

(7) When a variable is passed by _____ , a copy of the variable is passed.

(8) When a variable is passed by reference the keyword ___ must precede it in the parameter list.

Answers:

(1) Procedures, functions (2) parameter (3) nested

(4) level (5) identifiers (6) reference (7) value

(8) var

ADVANCED DATA TYPES

Array data type

Another cousin to the data types already explained is the array. Sometimes a large number of variables of a particular type are needed. If for example you required seventy two variables of the type char to represent a user's input character string from a file, you could declare them as previously explained. The disadvantage is obvious, as the effort would be time consuming. Furthermore, accessing the individual variables would be confusing, as each would have a different name.

There is a simple answer to this problem and it is the data type array. You may declare a variable as :

```
            variablename : array [1..n] of type;
or          variablename : array (.1..n.) of type;
```

Note - In TRS-80 Pascal, (. may be substituted for [, and .) for]

where type is any previously defined data type and n is the number of variables desired. For now we will concern ourselves with integer dimensions. Integer dimensions may be any positive or negative numbers such that the range of dimensions do not cause a storage overflow. This is a machine dependent constraint that varies among implementations. Thus we may declare :

```
VAR     line : array  [1..72] of char;
        varname:char;
```

To access a component of this array you would use a subscript denoting the numerical element. An example assignment might be varname := line[4]; . Varname would be set to the value of the fourth element in the array line.

Any array may be declared with the word PACKED as a prefix. The packed attribute tells the compiler to store the data elements as efficiently as possible. In Standard Pascal, you may not pass elements of packed structures by reference to procedures or functions, and packed elements may not be used as arguments in READ statements. In TRS-80 Pascal, there are no such restrictions.

Arrays

```
listing 9.1

PROGRAM onedimarray;
VAR
   stringl: PACKED ARRAY [1..72] OF CHAR;
BEGIN
   WRITELN('Enter command string');
   READLN(INPUT,stringl);
   WRITELN(OUTPUT,stringl);
   WRITELN('Program complete');
END.
```

If the I/O is directed to the terminal, the program will display the prompt: "Enter command string" . At this point the user may type up to seventy two characters of input, terminated with the return key. The input characters will be input to the array "stringl" left justified. If the input character string is less than seventy two characters in length, the remaining storage positions in the array will contain blanks. At this point the input message will be echoed to the terminal. In TRS-80 Pascal, an entire single dimension packed or nonpacked array of char may be input/output by a single read/write.

Arrays in Pascal may have multiple dimensions. Suppose that you had a number of input character strings as in the previous example, and it was desired to store every character string. A simple answer would be to declare the array line : array [1..5,1..72] of char; . This declaration is a Pascal short form for the declaration of array[1..5] of array[1..72]of char; .

If the data structure is a two dimensional array of char, then the read command will not input the entire array automatically, but instead requires that each individual sub-array be read in with a separate read statement.

Remembering that any single dimension array may be input by a single READ statement, leads to the following example array input sequence. Examine the following program.

listing 9.2

```
PROGRAM arrayIO;
VAR
   I          : INTEGER;
   stringl    : ARRAY [1..5,1..72] OF CHAR;
BEGIN
   FOR I:= 1 TO 5 DO
       BEGIN
       WRITELN(OUTPUT,'Enter command line  ',I);
       READLN(INPUT,stringl[I] );
       END;
END.
```

This program will prompt the user for five different command lines. In each case, the individual sub arrays are loaded into the array by the program.

Since individual array elements are of the type char, any operations that can be performed on a simple variable, of type char, may be performed on an array element. Remember also that arrays may be of any type such as boolean, integer or any user defined data types, including arrays. Arrays may have upper and lower bounds declared as constants in the declaration, and in fact, the name of most simple data types may be substituted for the bounds. The number of array elements in this case is determined by the number of elements in the data type.

User defined data types

The data types explained so far have been pre-defined. In Pascal, you may define new data types at will. These defined types have names chosen by the programmer and are declared in the TYPE section. Once declared, they may be used where predefined type names are allowed. This is a very powerful feature. Take for example the case where a programmer is manipulating an integer variable in Basic that may take on one of four values, 1..4. The numbers may represent the colors red, green, blue and orange. When the value is 1 : a message is written to the terminal saying that the color red is being processed, 2 : That the color green is being processed and so on. This is typically known as decoding information from a variable's value. Needless to say, when Basic programs get very long, it is difficult to determine their flow because of this decoding and encoding of information. A simpler way would be to declare a variable that could take on the value of red, green, blue and orange. Then tests could be performed to see if the value of the variable is red, etc. Program logic would be much clearer and easier to follow. In fact, this is exactly what the following program does.

listing 9.3

```
PROGRAM usertypes;
TYPE  difcolor = (red, green, blue, orange);
VAR
   color: difcolor;
BEGIN
   color := red;
   REPEAT
      CASE color of
         red    : WRITELN(OUTPUT,' The color is red');
         green  : WRITELN(OUTPUT,' The color is green');
         blue   : WRITELN(OUTPUT,' The color is blue');
         orange : WRITELN(OUTPUT,' The color is orange');
         END;
   color := succ(color);
   UNTIL( color = orange );
END.
```

Enumerated user defined types

Program 9.3 illustrates an enumerated user defined type, "difcolor" An enumerated type is where a list of possible variable values are given in the type declaration. The predefined function, "succ" is available in Pascal, and is a convenient way of incrementing a user defined variable type to the next possible value. In a simple program using an integer variable, this could be accomplished by adding one to the variable, but this would not make sense with a user defined type. User defined enumerated data types may not have their values written out. Program 9.3 gives an example of how that may be accomplished.

Subrange types

A variable may assume a value that is in a sub- interval of some other simple type. In this case, it may be declared to be a subrange type. For example, integer may represent all whole numbers between -32,768 and 32,767. In the type section, a subrange user defined type might be declared to be byte = 0..255 ; I.E.; any variable of the type byte may take on the value from 0 to 255 . The same operations may be performed on a subrange type that are applicable to the original type. Also a subrange type may be the subrange of any user defined simple type.

```
listing 9.4

Program subrange;
TYPE
   baddate           = 1900..1903;
   uppercaseletters  = 'A'..'Z';
   lowercaseletters  = 'a'..'z';
   digits            = '0'..'9';
   xaxis             = -100..100;
VAR
   testyear           : baddate;
   upperletter        : uppercaseletters;
   lowerletter        : lowercaseletters;
   digit              : digits;
BEGIN
END.
```

All of the above examples are valid subrange declarations. Named subrange types are very helpful when a programmer wants to clearly identify the data differences between specific variables to increase readability. Also, the storage required for a subrange variable is proportional to the interval it spans. This may be important when building large data structures to be implemented on microcomputers.

RECORD data types

So far, the only structured data type examined has been the array. The array is an excellent mechanism for storing large amounts of data of the same TYPE. For example, the series of text strings input from the terminal were efficiently stored using arrays of CHAR, and any individual character was easily accessible. However, it is often desired to keep variables of different data types grouped together. Take for example, a list of a business's customers along with vital information about each customer. Suppose that you desired to keep the following information about every customer:

```
Name
Customer category
Mailing address
Telephone number
Dollars spent in store
On catalog circulation list
```

This might represent a situation where the business would like to keep a data base updated. In languages like Basic, the only way to maintain this information would be multiple arrays containing encoded information. This is not the case in Pascal, as you may build a RECORD which can store all of the above information in a clear and concise format. Furthermore, you may declare an array to be of this user defined type.

Record data types

In Pascal, a RECORD is a predefined data structure which is composed of component variables. These component fields may be variables of any Pascal predefined, or user defined data types. The purpose of a record is to group variable information into logical entities, such that any particular component field may be operated on, or the entire record may be referenced as a whole. The following is an example of how the previous business record is declared in Pascal.

Listing 9.5

```
PROGRAM database;
TYPE
  custmrcategory = (business,individual);
custmrecord = RECORD
  custmrtype   : custmrcategory;
  address      : PACKED ARRAY[1..72] OF CHAR;
  telephone    : PACKED ARRAY[1..15] OF CHAR;
  expenditures : REAL;
  cataloglist  : BOOLEAN;
  END;
VAR
   custmr     : custmrecord;
   custmrlist : ARRAY[1..100] OF custmrecord;
   index      : INTEGER;
   ans        : CHAR;
PROCEDURE custmrinp( VAR custmr:custmrecord);
VAR custyp : CHAR;
BEGIN
  WRITELN('* Enter customer type: (business/individual)');
  READLN(custyp);
  IF(custyp='I')THEN
    custmr.custmrtype:=individual
  ELSE custmr.custmrtype:=business;
  WRITELN('* Enter address:');
  READLN(custmr.address);
  WRITELN('* Enter telephone number:');
  READLN(custmr.telephone);
  WRITELN('* Enter expenditure in dollars:');
  READLN(custmr.expenditures);
  WRITELN('* Want on catalog circulation list: (true/false)');
  READLN(custmr.cataloglist);
END;
BEGIN
  index:=0;
  ans:='N';
  WRITELN('** BUSINESS XYZ CUSTOMER RECORD PROGRAM **');
  WHILE (ans <> 'S') DO
    BEGIN
    index:=index+1;
    custmrinp(custmrlist[index]);
    WRITELN('* MORE CUSTOMERS (STOP/CONTINUE)');
    READLN(ans);
    END;
END.
```

The outer shell that must enclose record type declarations is of the form:

```
type name = RECORD
            END;
```

The component field declarations reside between the RECORD and END; . The field declarations are defined in the same way as the VAR section of the program. In program 9.5, the user defined record name is custmrecord. The component field declarations: custmrtype, address, telephone, expenditures, and cataloglist are defined exactly the same way as the program variables are in the VAR section. All the field components belong to the data type custmrecord. Since custmrecord is treated like any other user defined type, we may now declare a variable to be of type custmrecord in the program VAR section.

The difference between a record and other simple user defined data types is that there are component fields in a record that are really variables themselves. In example 9.5, the variable custmr is of a record type. When referring to custmr in expressions, to reference the entire record, you simply use the variable name, custmr. To access the component field, expenditures, you would prefix expenditures with the record variable name, custmr, separated with a '.' character. Example:

```
custmr.expenditures:= 99.95;
```

If another record named excustomer had been declared, the following would be a valid statement.

```
excustomer:=custmr;
```

In this case, all component fields in excustmr would be set to the component fields in custmr. Variables of type record, and their associated component fields, obey the same rules for use as all other typed variables.

The purpose of program 9.5 is to perform record I/O utilizing the predeclared text files input and output. Notice the read and write statements utilize record component fields as arguments. Read and write behave as though the component fields were variables declared in the VAR section. As with other variables, I/O may not be performed to a text file through a component field that is of a user defined enumerated type.

WITH statements

The use of records may often cause segments of the program that reference them to become long and tedious, because every time a component field is referenced, the record variable name must precede it. Accessing component fields may be simplified by using the WITH statement. Examine the following procedure, which could be included in program 9.5 .

Listing 9.6

```
PROCEDURE custmroutput(VAR custmr:custmrrecord);

BEGIN
   WRITELN('** CUSTOMER OUTPUT RECORD FOR BUSINESS XYY **');
   WITH custmr DO
        BEGIN
        IF(custmrtype=business)then
           WRITELN  ('Customer type    : Business')
        ELSE WRITELN('Customer type    : Individual');
        WRITELN     ('Address          : ',address);
        WRITELN     ('Telephone        : ',telephone);
        WRITELN     ('Expenditures     : ',expenditures);
        WRITE       ('Circulation list : ');
        IF (cataloglist)THEN WRITELN('Yes')
        ELSE WRITELN('No');
        END;
END;
```

The action of the WITH statement is to eliminate the normally required record variable name prefix when accessing component fields of that record. The scope of the WITH is one statement, which in this case is a compound statement.

File of TYPE

INPUT and OUTPUT are examples of TEXT files in Pascal. These FILE types have been used for all of the program examples so far. A TEXT file is TRS-80 Pascal predeclared to be a special file of char, with rules for performing I/O using INTEGER, REAL and BOOLEAN variables. In TRS-80 Pascal, there are extensions to allow for performing I/O using ARRAY variables in text files.

A FILE OF <any known type> may be declared in Pascal. Files of types other than text are primarily used for storing data which will be retrieved at some other time. For example, a FILE OF customerecord could be defined in the type section. (customerecord as defined in listing 9.5) A variable of type customerecord could be written to this file. The important thing to remember is that an entire record may be written (or read), by one I/O statement. Component fields of this record may not be read or written individually to a file of records. When I/O is performed with a FILE OF <any type except text>, no ASCII encoding or decoding of information takes place. Instead, the binary representation is used. This is not particularly useful when the I/O is directed to a terminal, but is effective for storing large amounts of information on disk media. The predeclared procedures WRITELN and READLN are not valid when performing I/O with a file of any type except TEXT, although read and write perform normally. The program in the appendix of this manual utilizes a FILE OF custmrecord for storing information in a data base. This is a typical use for a FILE OF TYPE.

(1) If a large number of variables of the same TYPE need to be declared, the _____ may be the correct data structure to use.

(2) Arrays in Pascal may have more than _____ dimension.

(3) New user defined _____ may be declared in Pascal programs.

(4) An _____ TYPE is defined by a list of identifiers given to be the different values allowable for a variable.

(5) A _____ TYPE is any user defined TYPE that is a sub-interval of another simple TYPE.

(6) A _____ TYPE is used to logically group together data of different types.

Answers:

(1) array (2) one (3) types (4) enumeration

(5) subrange (6) record

DYNAMIC DATA TYPE

All of the variable types discussed so far have been "static" in nature. This means that the size of data structures such as the array have to be defined before the program is compiled or executed. In program 9.5, the size of the array customer list has an upper bound of 100 entries. If more than 100 storage locations are needed to store the customer records, the array declaration has to be changed in the source program, and the source recompiled. In most popular micro-minicomputer Pascal implementations today, there are limits to the number of storage locations that may be declared in a program. This limitation is usually proportional to the size of the program in conjunction with the type and number of variable declarations. It is usually impractical due to these memory restrictions to declare arrays and other data structures to be larger than required. The static nature of variable declarations often create problems in some programming applications. Suppose for example, that in program 9.5 it was desired to keep a list of sales transactions for each customer attached to each customer record. This could be accomplished by declaring a component field of each customer record as being an array of transaction records. Then at any time you could access the sales transaction of every customer. This would require that the number of sales transactions per customer be limited to a preset number by the array declaration. It might be feasible to limit the number of customers to 100, but the number of transactions per customer might vary. There is a mechanism in Pascal to allow for dynamic variable allocation at program execution time. It is possible to request a new storage location for a variable by calling the Pascal pre-defined procedure NEW.

Procedure NEW

By calling the procedure NEW, it is possible to get a pointer to a memory storage location that is the proper size for the argument variable. It is important to remember that the same limitations on the amount of memory available still apply, however dynamic allocation of memory allows for better utilization of space. The variable used as an argument for the NEW procedure call must have been declared in the VAR section. It must be declared as a TYPE that is a pointer to the actual data type. An example of a pointer data type declared in the TYPE section is as follows:

Listing 10.1

```
TYPE
     trxptr = ^trxrec;

trxrec = RECORD
   nexttrx: trxptr;
   invoicenumber   : INTEGER;
   date            : ARRAY [1..10] OF CHAR;
   transprice      : REAL;
   partnumberlist : ARRAY [1..10] OF CHAR;
   END;

VAR
   trx : trxptr;
```

In the example program segment, the variable trx is of type trxptr. In the type section, trxptr is defined to be a pointer to " trxrec " . The character " ^ " denotes a pointer in Pascal. Therefore, the variable trx is a pointer to a storage location in memory of the size required to store the RECORD trxrec. This storage location may be requested anytime during program execution as opposed to program startup. Pointer types to large data structures may be declared in a program with minimum memory space penalty until the procedure NEW is called during program execution. Notice at the "^trxrec" point in the type declaration, trxrec has not been defined. In Pascal, declaring a pointer to an as yet undefined type is valid.

The following program segment illustrates a few simple methods of using pointer variable types.

```
 Listing 10.2

PROGRAM dynamic;

(* TYPE declaration section from listing 10.1 *)

VAR
  trx      : trxptr;
   nexttrx : trxptr;
BEGIN
   NEW(trx);
   trx^.invoicenumber:=2345;
   trx^.transprice   :=99.95;
   nexttrx:=trx;
   WRITELN('* Transaction invoicenumber : ',
         trx^.invoicenumber);
   WRITELN('* Transaction price         : ',
         trx.transprice);
   DISPOSE(trx);
END.
```

If the pointer itself is being referenced, just the variable name is used. In the example, the pointer variable nexttrx is set to the value of trx. When referring to the contents of the storage location, an " ^ " follows the variable name. "trx^.transprice" refers to the value of the component field stored at that location. These basics of pointer data type manipulation are used to build "linked lists" . A linked list is a chained list of dynamic storage areas.

Notice the procedure call to DISPOSE. The purpose of DISPOSE is to release the storage area acquired in the NEW call. After the DISPOSE, the data stored at the dynamic memory location is effectively lost. This is an important feature of Pascal. Careful use of NEW and DISPOSE can result in programs that dynamically grow and contract in memory size as needed, and efficiently manage the computer resources.

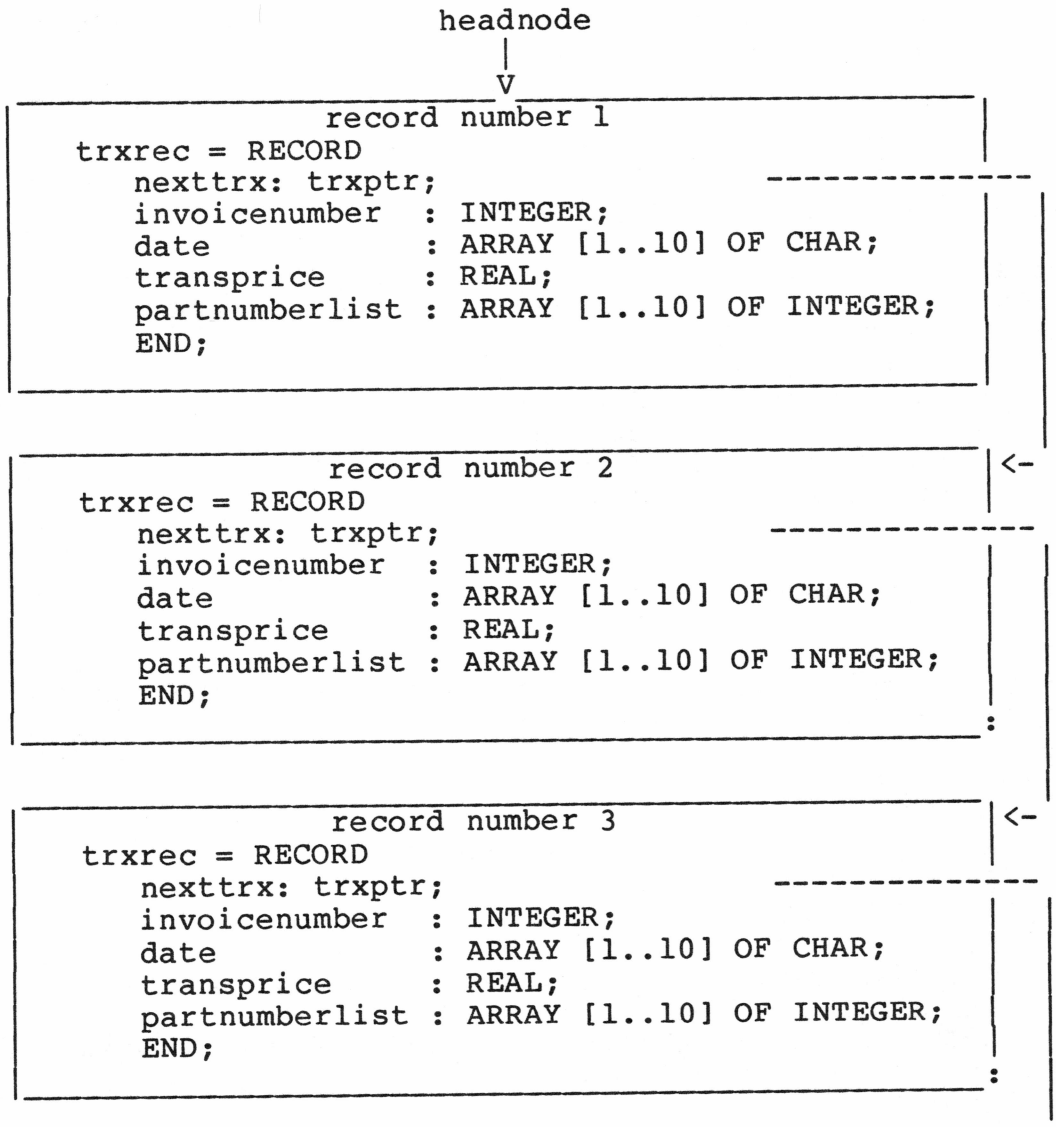
LINKED LIST

A linked list is a programming technique that chains together a series of variables. A thorough discussion of linked list processing would entail several chapters, and is really a topic for a data structures book. It will be covered briefly here because it is integral to discussions about dynamic memory management.

In example 10.1 , the data type trxrec has a component field which is a pointer to a storage area of the same type as itself. A pointer to another record node may be stored in this field. In the record pointed to, a pointer to another record node could be stored, and so on. In this way, a series of record nodes may be linked together. The following diagram will help to visualize this list.

Listing 10.3

```
VAR
   headnode : trxptr;
```



The variable headnode is a pointer variable declared in the VAR section of the program. At some point in the program, a NEW procedure call could be made with headnode as its argument. Headnode would now be a pointer to the start of the list. Notice the word NIL at the end of the list. NIL is a reserved word in Pascal. This simply sets the pointer to an initialized value that may be tested for in looping statements. A word of caution when using pointers in Pascal. If a pointer variable has been declared, but not set to any value, there is no guarantee of its value. It will not necessarily be set to NIL . Most Pascal implementations do not perform a runtime check for uninitialized values. Use of uninitialized pointers can lead to the program writing over itself in memory with execution becoming unpredictable. These kinds of programming errors will not show up at compile time, and can be extremely hard to find during program execution. The following segment illustrates how list 10.3 could be built.

Listing 10.4

```
PROGRAM linkedlist(input,output);
TYPE
   trxptr = ^trxrec;
   textline       =  PACKED ARRAY [1..10] OF CHAR;
   trxrec = RECORD
      nexttrx: trxptr;
      invoicenumber  : INTEGER;
      date           : textline;
      transprice     : REAL;
      partnumberlist : textline;
      END;
VAR
  headnode,transnode:trxptr;
                  I : INTEGER;
PROCEDURE readtrx(VAR trx:trxrec);
  (* The purpose of this routine is to prompt the user for *)
  (* the purchaser's trx record                            *)
BEGIN
  WITH trx DO
      BEGIN
      WRITELN('ENTER INVOICE NUMBER:');
      READLN(invoicenumber);
      WRITELN('ENTER DATE:');
      READLN(date);
      WRITELN('ENTER TOTAL PURCHASE PRICE:');
      READLN(transprice);
      WRITELN('ENTER PARTNUMBER(S) SEPARATED BY COMMAS:');
      READLN(partnumberlist);
      END;
END;   (*readtrx*)
```

Listing 10.5 (continuation 10.4)

```
PROCEDURE writetrx(VAR trx:trxrec);
(* The purpose of this routine is to write the purchaser *)
(* trx entry                                              *)
VAR   I:INTEGER;
BEGIN
   WITH trx DO
      BEGIN
      FOR I:= 1 TO 35 DO WRITE('*');
      WRITELN;
      WRITELN('INVOICE NUMBER       : ',invoicenumber);
      WRITELN('DATE                 : ',date);
      WRITELN('TOTAL PURCHASE PRICE : ',transprice:10);
      WRITELN('PART NUMBER LIST     : ',partnumberlist);
      FOR I:=1 TO 35 DO WRITE('*');
      WRITELN;
      WRITELN;
      END;
END;  (*PROCEDURE writetrx*)

PROCEDURE listrxs( temptr : trxptr );
(* the purpose of this procedure is to traverse the linked*)
(* list attached to the argument pointer, writing the     *)
(* values of the trx records                              *)
VAR
  loctrx : trxrec;
BEGIN
  (* traverse trx linked list,writing trxs *)
WHILE (temptr <> NIL) DO
  BEGIN
    (* load the contents of localtrx with the *)
    (* contents of temptr                     *)
  loctrx:=temptr^;
  writetrx(loctrx);
    (* set temptr to the next node in the linked list  *)
  temptr := temptr^.nexttrx
  END;
END;   (*listransactions*)
```

Listing 10.6 (continuation 10.5)

```
BEGIN     (* begin main program linkedlist *)
  (* initialize pointer that will always reflect the *)
  (* beginning of the list.                                          *)
  (* this will set the end of the list to NIL during the first       *)
  (* pass through the FOR loop                                       *)
headnode := NIL;
  (* read 3 trxs and link each new one to the beginning *)
  (* of the list                                                     *)
FOR I := 1 to 3 DO
  BEGIN
  NEW(transnode);
    (* insert the newnode in front of the old headnode *)
    (* link to the old headnode                        *)
  transnode^.nexttrx := headnode;
    (* make the newnode the new headnode               *)
  headnode := transnode;
    (* load the actual data into the fields of the new node *)
  readtrx(transnode^);
  END;
  (* list all trxs entered *)
listrxs(headnode);
END.  (*main program*)
```

Sets

Sets in Pascal have the same meaning as they do in the normal mathematical sense. If a group of objects are declared in set A, and a group of objects are declared in set B, a number of operations may be performed on these sets such as :

(1) Membership and relational testing
(2) Set arithmetic (union,intersection,difference)

In the case of Pascal, the objects are simply data values. These data values may be Pascal predefined or user defined. An example would be a SET OF CHAR, or a SET OF digits where digits is a user defined subrange type of CHAR. Testing could be performed to see if the SET OF digits is in the SET OF CHAR if desired. The method of declaring set variables is :

```
VAR    A,B : SET OF <type> ;
```

This means that A and B may contain from one to all of the data values declared by the type, however its membership is undefined until it is initialized like any other variable. In the body of the program, a set may be initialized to empty by :

```
A := [];
```

Membership testing

Once the set variables are initialized, a series of BOOLEAN relational tests may be performed. The relational operators are as follows:

set1 = set2	Set equality- If (all members of first set are in the second set and all members of second set are in the first set) : returns true.
set1 <= set2	Subset- If (all members of first set are in the second set) : returns true.
set1 >= set2	Superset- If (all members of second set are in the first set) :returns true.

set1 <> set2	Set inequality- If all members of first set are in second set, and all members of second set are in first set : returns false.

Individual element membership may be tested by using the IN operator. If a variable had been declared of the same type as the base set type, the IN operator may be used to check for set membership. An example would be:

Listing 12.1

```
TYPE
  DIGITS = '0'..'9';
VAR
        DIGIT  : SET OF DIGITS;
        D      : CHAR;

 BEGIN
   D:='a';
   DIGIT:=['0'..'9'];
   IF(D IN DIGIT)THEN DO (*action*);
   IF(D='0')OR(D='1')OR(D='2')OR(D='3')OR(D='4')OR(D='5')
     OR(D='6')OR(D='7')OR(D='8')OR(D='9') THEN DO (*ACTION*)
```

The two IF statements in the above program segment are equivalent. Notice that the equivalent IF statement using sets is a more concise and readable statement. This represents a simple use for sets for the average programmer.

Set arithmetic

There are three set operators in Pascal. Each requires two arguments. Arguments should be sets of the same base type, and the result will be of the same type. The operators are:

A + B	Gives the union of A and B
A * B	Gives the intersection of A and B
A - B	Gives the difference of A and B.

The following segment program illustrates set operator use.

```
LISTING 12.2

PROGRAM TESTSET;
VAR
  DIGITS,LETTERS,LOWERCASE,UPPERCASE : SET OF CHAR;
  ALPHANUMERIC,ALPHA                 : SET OF CHAR;
  D : CHAR;
BEGIN
  D:='1';
  DIGITS:=['0'..'9'];
  LOWERCASE:=['a'..'z'];
  UPPERCASE:=['A'..'Z'];
  LETTERS:=LOWERCASE + UPPERCASE;
  ALPHANUMERIC:=LETTERS + DIGITS;
  ALPHA:=ALPHANUMERIC - DIGITS;
  IF ( D IN ALPHANUMERIC * DIGITS ) THEN
    WRITELN('PUNT');
END.
```

On diskette there is a file named DATABASE/PCL . This source program ties all of the previous program segments in chapters 9 and 10 together, to build a program that will build a data base for business customers. This is not intended to be a comprehensive program, but can serve as a starting point for an expansion. This program requires approximately 15K of stack to RUN or execute. Once compiled, it may be executed by typing:

```
RUN DATABASE 15K
```

The number of customers allowed in the data base array is set by the constant "maxarray" , and may be changed to reflect local memory restrictions. Customer transactions are linked to each customer record by dynamic management of linked lists. Customer records are kept on a separate file from the transactions in order to simplify rebuilding of the linked lists when loading an existing data base. The size of the data base accessible during a program invocation is limited by the available memory, as the entire data base is loaded into memory for operations. Large data bases may be accessed by partitioning the data base between files and running the program multiple times.

LANGUAGE REFERENCE MANUAL

TABLE OF CONTENTS

Notation and Terminology.....5

Chapter 1

Program Elements.....7
A. Identifiers.....7
B. Numbers.....8
C. Strings.....9
D. Reserved Words.....9
E. Special Symbols.....10
F. Comments.....10
G. The Semicolon.....11

Chapter 2

Program Structure.....12
A. Block Headings.....13
1. PROGRAM Heading.....13
2. PROCEDURE Heading.....13
3. FUNCTION Heading.....15
B. Block Parts.....16
1. LABEL Declarations.....17
2. CONSTant Definitions.....18
3. TYPE Definitions.....19
4. VARiable Declarations.....20
5. COMMON Declarations.....20
6. ACCESS Declarations.....21
7. PROCEDURE and FUNCTION Declarations.....22
8. Statement Body.....23

Chapter 3

Simple Data Types.....24
A. Ordinal Types.....24
1. INTEGER.....24
2. CHAR.....25
3. BOOLEAN.....25
4. Enumeration.....26
5. Subrange.....27
B. REAL Type.....27

Chapter 4

Structured Data Types.....28
A. ARRAY.....28
B. SET.....29
C. FILE.....32
1. Predefined Type TEXT.....33
D. RECORD.....34

Chapter 5

Pointer Data Type.....40

Chapter 6

Operators.....44
A. Arithmetic.....44
B. Relational.....45
C. Boolean.....46
D. Precedence.....47
E. Type Transfer.....48

Chapter 7

Expressions.....49

Chapter 8

Statements.....53
A. Assignment.....54
B. Compound.....55
C. Repetitive.....55
1. FOR.....56
2. WHILE.....57
3. REPEAT.....57
D. Conditional.....58
1. IF.....58
2. CASE.....60
E. WITH.....61
F. GOTO.....63
G. Procedure.....63

Chapter 9

Procedures and Functions.....65
A. Scope Rules.....66
B. FORWARD.....68
C. EXTERNAL.....69
D. Recursion.....71
E. Predeclared.....72

Chapter 10

Input and Output....76
A. RESET....77
B. REWRITE....78
C. READ....79
D. WRITE....81
E. READLN....84
F. WRITELN....85
G. CLOSE....86
H. PAGE....87
I. MESSAGE....87

Appendix....88
A. Compiler Options....88
B. Error Messages....98
C. ASCII Character Set....102
D. Differences from Standard....105
1. Omissions....105
2. Extensions....105
3. Other Implementation Characteristics....107
E. Strings....108
F. Get and Put I/O procedures....112
G. Using files in structured variables....115
H. Using global variables in externals....118
I. Using common variables....119

FOREWORD

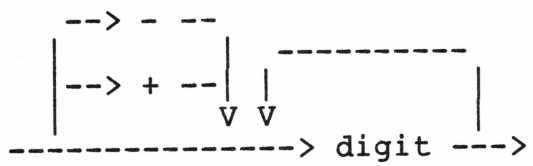
This manual assumes that the reader is already somewhat familiar with the Pascal language. It is organized to be used as a reference manual. As such, the chapters group related topics in order to make them easier to find. The result of this is that the manual does not follow a progression of discussion which is well suited as a teacher of the Pascal language. It is suggested that you first read the Pascal Tutorial if this is your first experience with the language.

The description of any programming language involves both the syntax and the semantics of the language. The syntax refers to the arrangement of program elements into a form which the compiler can understand. The semantics refers to the meaning that the compiler associates with a particular arrangement of the program elements. The semantics of a language can be explained with words but the syntax is best explained through the use of diagrams.

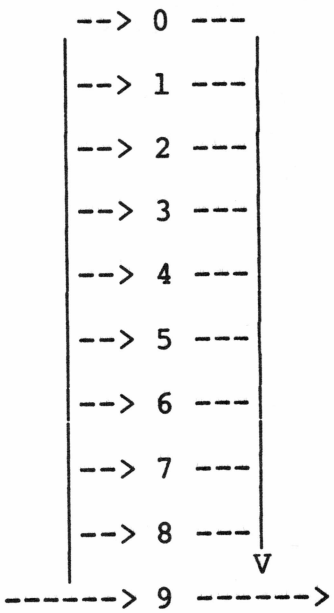
The syntax diagrams used throughout this manual describe the legal syntax of a program. Each diagram has an entering and an exiting point which is denoted by an arrow. Starting with the arrow entering a diagram, the legal syntax can be determined by tracing a path which follows the directions indicated by the arrows until the exiting arrow is reached. Most diagrams have a multiple number of paths from starting point to ending point. All paths describe a syntactically correct form.

The following are sample syntax diagrams which describe the syntax of an integer.

Syntax of an integer:



Syntax of a digit:



The syntax diagram for an integer says that an integer is a concatenation of one or more digits which is optionally preceded by a plus or minus sign. Entering the diagram, you have 3 possible paths from which to choose. One path leads directly to "digit", one leads to "+", and one leads to "-". The paths from both "+" and "-" then lead to "digit". Passing through "digit", you have the option of exiting the diagram or following the arrow which leads back to the beginning of "digit". From this point, you pass through digit again and optionally exit or return for another pass. Thus, an integer may consist of one or more digits.

The second syntax diagram describes the correct forms of a digit. Entering the diagram, you have ten possible paths from which to choose. All paths lead to a single character, each of which is a legal digit. Choosing a path, you follow it through a character and end up at the exiting arrow. At this point, there is no alternative but to exit the diagram. No other paths are available. Some examples of integers then are 10, +963, and -75.

In the diagrams used in this manual, upper case character strings denote reserved words that must be present in the form shown. Lower case character strings denote the parts of the syntax where many legal forms exist. For example, the word integer in a diagram in lower case letters represents any legal integer. The word INTEGER in uppercase letters represents a reserved word of the language.

In some cases, abbreviations are used to shorten a diagram. For example, id is used in place of identifier. Also, expr is used in place of expression. A few other abbreviations may occur but where used, their meaning should be apparent from the surrounding text.

PROGRAM ELEMENTS

The elements of a program consist of the entities (identifiers, numbers, strings, reserved words, and special symbols) which are composed from a character set. The ASCII character set is the most often used and is listed in the appendix.

A. Identifiers

An identifier serves to denote the program name, a constant, a type, a variable, a procedure, or a function. It consists of a letter followed by combinations of zero or more of the following characters:

(the 26 letters of the alphabet in lower or upper case, the digits 0 through 9, the character $, the character _).

Note:
no distinction is made between upper and lower case letters in identifiers. The two identifiers, Apple and apple, are considered identical.

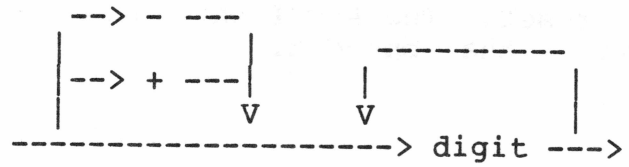
The length of an identifier is arbitrary but only the first 8 characters are significant. For example, the identifiers A2345678 and A23456789 would to the compiler be identical because it discards all characters past the eighth character. Therefore, care should be taken to make identifiers eight characters unique. It should also be noted that an identifier cannot contain embedded blanks or span a line boundary.

Examples: Factor$ DEPARTMENT A Div_10 B12345678$_

B. Numbers

Numbers are integer or real constants. Integers are allocated sixteen bits of storage which imposes a size limitation. The range for an integer is -32768 to +32767.

Syntax of integer numbers:

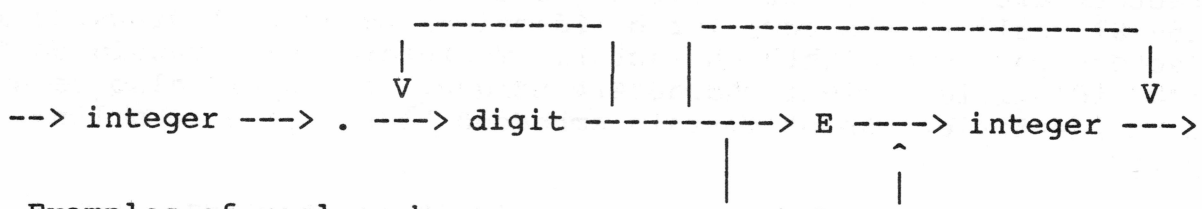


Examples of integer numbers:

30 -28934 0 32739

Real numbers are represented in either exponential or fixed point form. The fixed point form consists of an integer part followed by a decimal point and a fractional part. The exponential form consists of a fixed point part followed by an exponent part. The exponent part is a multiplier. The value of a real number in exponential form is the fixed point part times (10 raised to the exponent part). (See the System Implementation Manual for the size, range, and accuracy of real numbers).

Syntax of real numbers:



Examples of real numbers:

Fixed point form:

50.0 -100000.0 345.22452

Exponential form:

0.239E3 -4.5921E-2 876.0E+33 193.27D-3

0.239E3 is equivalent to 239.0
-4.5921E-2 is equivalent to -0.045921

NOTE: Using D instead of E in exponential form represents a double precision real number.

C. Strings

Strings are sequences of characters enclosed by single quote marks. A string consisting of a single character is a constant of the type CHAR. Strings consisting of n characters, where n is greater than one, are constants of the type PACKED ARRAY[1..n] OF CHAR. If a string is to contain a single quote mark, it must appear twice in the sequence.

Examples: 'ABC' '12"QZW' 'BEGIN ' '***' ' %'

The string consisting of the single character ' is represented as ''''.

Characters in strings can also be denoted by hexadecimal numbers. A hexadecimal number is composed from the characters 0 through 9 and A through F. (See the ASCII character set in the appendix). The character # followed by 2 hexadecimal characters represents a single ascii character. The character represented is the one whose ordinal position in the character set corresponds to the hexadecimal number specified. This feature provides a mechanism for representing nonprintable characters. A consequence of giving the character # a special meaning is that it must appear twice in a string just as the character ' must when the character itself is to be made a part of the string. A string consisting of the single character # then is represented by '##'.

Examples of hexadecimal character representation in strings:

```
'#30'   is equivalent to '0'
'D#4FG' is equivalent to 'DOG'
'#00'   corresponds to the nonprintable null character
'A#B'   is illegal
```

D. Reserved Words

The following list of words are keywords and have special meaning in a program. They may not be used as identifiers.

AND	DOWNTO	IF	OR	THEN
ARRAY	ELSE	IN	PACKED	TO
BEGIN	END	LABEL	PROCEDURE	TYPE
CASE	FILE	MOD	PROGRAM	UNTIL
CONST	FOR	NIL	RECORD	VAR
DIV	FUNCTION	NOT	REPEAT	WHILE
DO	GOTO	OF	SET	WITH

E. Special Symbols

The special symbols are used as operators and delimiters in a program. Because character sets vary from system to system, alternate representations are provided for some of the symbols.

Symbols with only one representation:

```
+     -     *     /
=     <>    <     <=    >=    >
(     )     '     :=    .     ,
;     :     #     ::
```

Symbols with alternate representations:

symbol	alternate
{	(*
}	*)
^	@
[	(.
]	.)

F. Comments

Comments can be used in a program for documentation purposes. The compiler generates no code for comments. The symbol { denotes the beginning of a comment while the symbol } denotes the end. All characters in between are ignored by the compiler. As shown above, the symbol { may be replaced by the symbol (* and the symbol } may be replaced by the symbol *).

Examples:
```
{this is a comment}
(*This is a comment
  that spans more than one line*)
```

Note: Comments may not be nested. The following will generate an error:

```
(*outer (*inner level*) level*)
```

G. The Semicolon

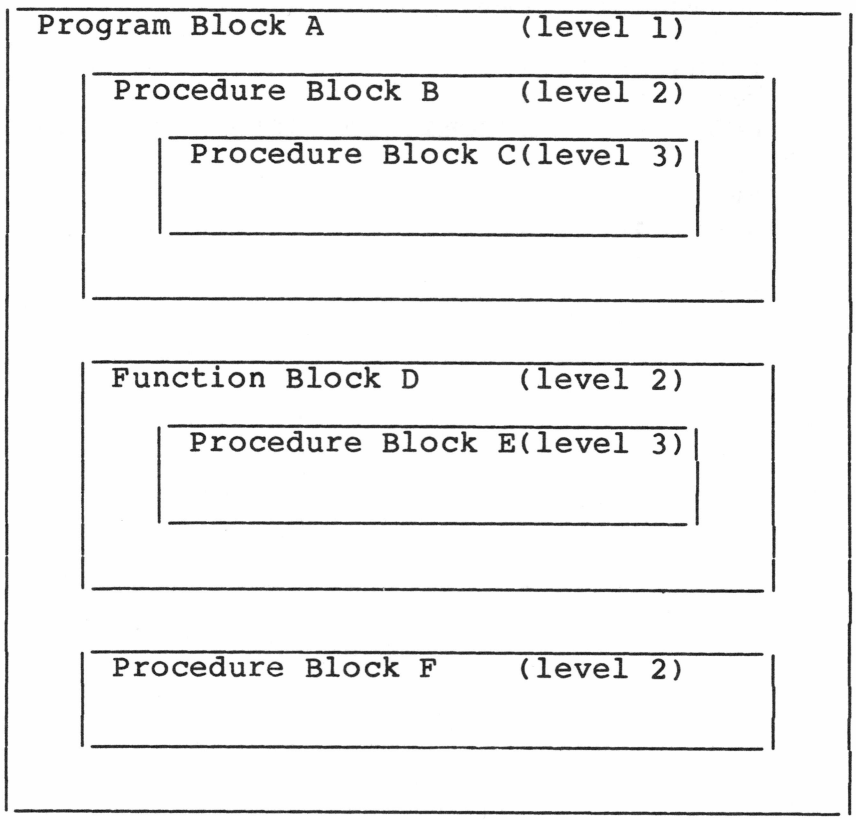
The semicolon is used extensively in the Pascal language. Its purpose is to separate the individual components of a program. For example, block headings must be separated from block parts, block parts must be separated from one another, and individual definitions, declarations, and statements within the block parts must be separated. In general, they may be used freely throughout the program. However, care should be taken not to include a semicolon in the middle of a statement. This is a common source for error when using the IF statement with one or more ELSE clauses. Since the ELSE clauses are a part of the IF statement, they must not be separated from it by a semicolon. An ELSE keyword should never be preceded by a semicolon.

example use of semicolons in an IF statement:

```
IF time > 12 THEN
  BEGIN
  alpha := 'e';
  beta  := 'f';
  END          (*semicolon here would cause an error*)
ELSE
  BEGIN
  alpha := 'g';
  beta  := 'h';
  END;
```

PROGRAM STRUCTURE

Pascal is a block structured language. This means that a program is constructed in a block like manner. At a minimum, a program consists of one block. More blocks are created through the use of procedures and/or functions by placing them inside this outermost "program block". The term for this process is called nesting. The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. This provides an appropriate tool for describing scope rules which are discussed in chapter 9. The block structured organization of a program can be represented pictorially by the following example:



A program then consists of at least one block, the program block, and optionally it contains procedure and/or function blocks which are nested within.

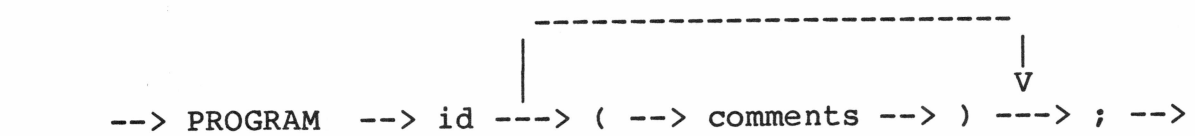
A. Block Headings

The purpose of the block heading is to give the block a name and in the case of procedure or function blocks, to define any parameters to be passed to the block. There are three types of blocks: the program block, the procedure block, and the function block. There is only one program block, the outermost block of the program, while there may be any number of procedure and function blocks. Each of the three types of blocks has a different heading. (Procedures and functions are discussed further in chapter 9)

A.1 The Program Heading

The program heading must be the first non-comment in a program. Its purpose is to signal the start of the program and to give the program a name. Characters inside the parentheses are ignored by the compiler.

Syntax of the program heading:



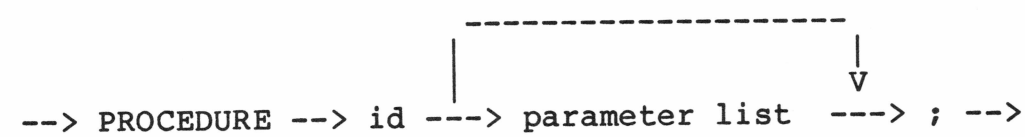
Example program headings:

```
PROGRAM lander;        PROGRAM taxes(computes income tax);
```

A.2 The Procedure Heading

The procedure heading signals the start of a procedure block. It gives the procedure a name and defines the parameters to be passed to it.

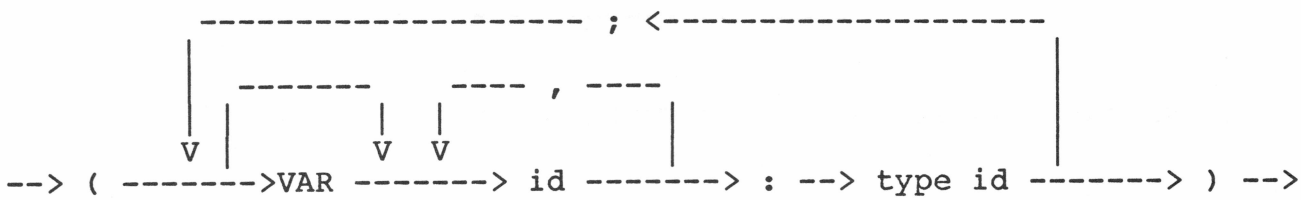
Syntax of the procedure heading is:



The parameter list declares the variables which are used to pass data into and out of a procedure. The variables are called formal parameters. The procedure statement which activates (or calls) a procedure has a corresponding list of parameters which are the actual parameters. The actual parameters must match the formal parameters in order and in type. However, their names need not be the same.

There are two different kinds of formal parameters, pass by value or pass by reference. A formal pass by value parameter causes its corresponding actual parameter to be copied to another location and then the formal parameter references the copied value. Therefore, changing the value of the formal parameter inside the procedure does not change the value of the corresponding actual parameter. In contrast, a formal pass by reference parameter is passed the address of the corresponding actual parameter. The formal parameter references the same location as the actual parameter. Therefore, changing the value of the formal parameter also causes the value of the actual parameter to be changed. Variable declarations in the parameter list which are preceded by the keyword VAR are pass by reference parameters while the absence of the keyword represents pass by value.

Syntax of the parameter list is:



Example procedure headings:

```
PROCEDURE out;

PROCEDURE cpu(pc : INTEGER);

PROCEDURE delete(VAR i,j :INTEGER; ch :CHAR; VAR x :REAL);
```

In procedure delete above, i and j are integers which are passed by reference, ch is a character which is passed by value, and x is a real which is passed by reference.

As a general rule, pass by value parameters should be used to prevent side affects. However, sometimes side affects are necessary. That is, sometimes you need a change in the value of a formal parameter to also change the value of its corresponding actual parameter. In such a case, pass by reference must be used. Also, when passing large data structures such as arrays, pass by reference should be used. This speeds execution and saves memory because a pointer to the structure is passed rather than copying the whole structure to another location.

A.3 The Function Heading

The function heading signals the start of a function block. It gives the function a name and defines the parameters to be passed to it. Unlike a procedure, a function has a type associated with it. Functions, like variables, are assigned values. A function is referenced by an expression and its value then substituted into the expression.

Syntax of the function heading:

The parameter list has the same form as the parameter list for a procedure discussed on the previous page.

Example function headings:

```
FUNCTION number : REAL;

FUNCTION nextstate(currentstate : INTEGER) : INTEGER;
```

The function "number" is a real valued function which has no parameters. The function "nextstate" is an integer valued function which has one parameter, also of type INTEGER. In each function, a value should be assigned to the function name. For example, number:=5.3 and nextstate:=currentstate + 1 could appear inside each of the respective functions to define values for them.

Note : a function may be an ordinal type or the type REAL only.

B. Block Parts

A Block is composed from the following list of parts.

1. the label declarations
2. the constant definitions
3. the type definitions
4. the variable declarations
5. the common declarations
6. the access declarations
7. the procedure and function declarations
8. the statement body

The label declarations are used to declare statement labels which can be used for branching. The constant definitions are used to give names to numbers or strings which are constants. Constant names are assigned values at compile time. Type definitions are used to create and give names to data types which are not predefined. Variable declarations are used to associate variable names to specific data types. A type defines the kind of data that can be stored in a variable. It also defines the amount of storage required for the variable. Variables are assigned values at run time. Common declarations are used in the same manner as the variable declarations to associate variable names to specific data types, but common variables have a special property. Storage space for common variables is created statically rather than dynamically. This means that when a block terminates, the common variables declared in it do not become undefined. Access declarations are used to enable a block to access a common variable. Procedures and functions are used for modularity. They provide the mechanism for segmenting a block into subblocks. The statement body contains the program statements which describe the actions to be taken on data as well as the order in which the actions take place.

A block does not have to include all eight parts described above. At a minimum, a block must include the two keywords BEGIN and END which bracket the statement body. The following is an example of a minimum complete program. It contains only the program block which is composed of only the heading and a null statement body.

```
PROGRAM donothing;
BEGIN           (*The statement body contains no statements*)
END.
```

The order in which the eight parts appear in a block is as follows: The first six parts may be arranged in any order. The only requirement is that an identifier be defined before it is used. For example, a particular type definition must textually precede a variable declaration of that type. The only exception to this is the definition of pointer types which are discussed in chapter 5. It is also worth noting that there may be more than one of a particular part. For example, there could be two separate type definition parts. The procedure and function declarations follow any use of the first six parts. The statement body then follows the last procedure or function declaration.

B.1 The Label Declarations

Label declarations are used in conjunction with the GOTO statement. A label declaration defines a label which can then be used to label a statement. A GOTO statement can then reference the label causing a branch to the statement which is prefixed by the corresponding label. The label declaration part is signaled by the keyword LABEL.

Syntax of the label declaration part:

Note:
A label must be declared in the same block in which a GOTO statement which references it appears. Branching outside a block is not allowed. Also, all declared labels must appear somewhere in the statement body.

Example label declaration part:

```
LABEL 100, 200, 300, 400, 500, 1000 ;
```

Syntax of labeled statement:

```
---> label --> : --> statement --->
```

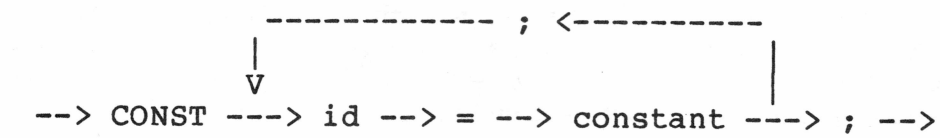
Example labeled statements:

```
100: x:=47;
200: IF x > 500 THEN GOTO 100;
```

B.2 The Constant Definitions

The constant definitions are used to associate identifiers with values which do not change. A constant identifier is assigned a value at compile time and this value can not be changed. This means that a constant identifier cannot have its value changed by an assignment statement. The use of constant identifiers increases program readability because meaningful names can be used in the place of actual values. The values which can be assigned to constant identifiers are numbers, strings, or other identifiers which are constants. This includes identifiers which are members of an enumeration. The start of the constant definition part is signaled by the keyword CONST.

Syntax of the constant definition part:



Example constant definition part:

```
CONST    low=32;  high=88;  pi=3.14159;
         speedoflight=299792.0;  separator='---------';
         positive=10;  negative=-positive;
         keydefinition=#61;
```

Note: Integer constants may also be expressed in hexadecimal by preceding the value with the #

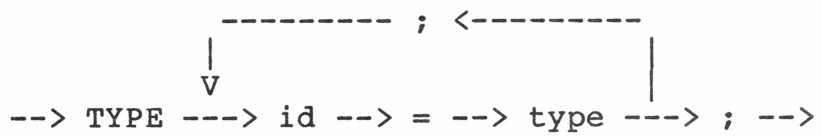
There is a predefined constant MAXINT which is defined to be equal to the largest positive value an integer can take.

B.3 The Type Definitions

Type definitions are used to create new data types. A type definition associates a name with a user defined simple or structured data type. The name can then be used in a variable declaration to specify the type of the variable. Although a variable can declare its type directly in the variable declaration part, it is nice and sometimes necessary to have a name associated with a user defined type. Type definitions are especially useful when using structured types whose definitions are long and when more than one variable in the program is to be declared of that type. Associating a name to the type means that the type must be defined only once. In some cases, type definitions are necessary. If comparisons are to be made between two variables of a user defined type, then the variables must be declared as the same type. Defining the type for each variable separately in a variable declaration part will not work. Although the variable declarations will look the same, the compiler will view them as variables of two separate types. Also, declarations of variables in the parameter list of a procedure or function must be to named types. For example, if an array is to be passed as a parameter, the array must be defined in a type definition and then the formal parameter declared as that type.

The type definitions part is signaled by the keyword TYPE.

Syntax of the type definition part:



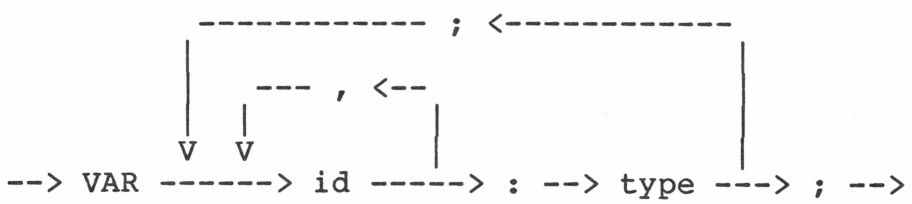
Example type definition part:

```
TYPE     colors = (red,blue,green,orange,purple);
         weekdays = (sunday,monday,tuesday,wednesday,
                     thursday,friday,saturday);
         workdays = monday..friday;
         daysofmonth = 1..31;
         letters = 'A'..'Z';
         list = ARRAY [0..25] OF CHAR;
         customer = RECORD
                    name    : PACKED ARRAY[1..20] OF CHAR;
                    address : PACKED ARRAY[1..40] OF CHAR;
                    END;
```

B.4 Variable Declarations

All variables in a program must be declared before they are used. This is done by associating the variable name with a type. The type can be the name of a predefined simple data type, the name of a user defined type which has previously been defined in a type definition part, or the type can be defined directly. The start of the variable declaration part is signaled with the keyword VAR.

Syntax of the variable declaration part:



Example variable declaration part:

```
VAR       x,y,z          : REAL;
          ok             : BOOLEAN;
          i,j,k          : INTEGER;
          fruit          : colors;
          alpha,beta     : CHAR;
          characters     : list;
          mark           : ARRAY [1..30] OF INTEGER;
          byte           : 0..255;
          months         : (jan,feb,mar,apr,may,jun,jul,
                            aug,sep,oct,nov,dec);
          account        : RECORD
                           number, date : INTEGER;
                           END;
```

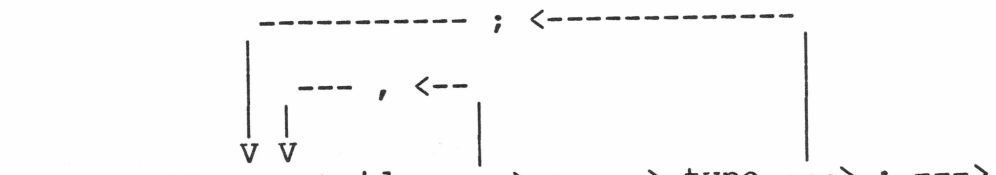
B.5 Common Declarations

The common declaration part is used in the same manner as the variable declaration part to associate a variable name with a specific data type. However, declaring a variable in the common declaration part gives it a special property. Normally, storage space for variables is allocated dynamically. This means that the variables declared in a block are allocated storage space when the block is activated and the space is freed when the block terminates. Therefore, the local variables of a block become undefined when the block terminates. In contrast, common variables are allocated storage space statically at compile time. This means that the common variables of a block retain their defined values even after the block terminates.

Common variables are scoped similar to normal variables. However, only one storage location is reserved for each common variable name. Therefore, a common variable declared locally within a procedure will reference the same location as a common variable of the same name declared anywhere else in a program.

A common variable cannot be accessed in a block unless its name appears in an access declaration of the same block. This feature is useful for controlling access to global variables, providing protection and better documentation of where global variables are used. Another very valuable use for common variables is in external procedures. A procedure which is often used by many separate programs can be compiled separately and linked to the programs that use it. In the case where the procedure must retain information between activations, such as cursor position in a graphics procedure, common variables may be used to prevent the need for global variables.

Syntax of the common declaration part:



Example common declaration part:

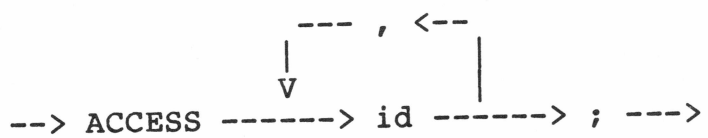
```
COMMON          cursorx , cursory  : INTEGER;
```

(see the appendix for an example using commons)

B.6 Access Declarations

Access declarations are used in conjunction with common variables. No common variable can be referenced unless its name appears in an access declaration of the block which references it. The order in which common variable names appear in an access declaration is arbitrary.

Syntax of the access declarations part:



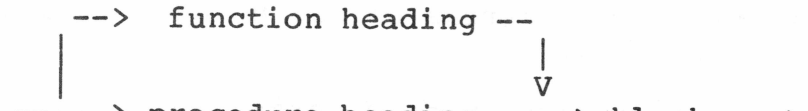
Example access declaration part:

```
ACCESS          cursorx , cursory;
```

B.7 Procedure and Function Declarations

Procedure and function declarations create new blocks. Each declaration forms a complete new block composed from the block parts discussed earlier. A procedure declaration consists of a procedure heading followed by a block. A function declaration consists of a function heading followed by a block. Procedure and function declarations form subblocks within the block in which they appear. Procedure and function declarations are discussed more fully in chapter 9.

Syntax of procedure or function declaration:



Example procedure declaration:

```
PROCEDURE getvalue(first,last :INTEGER; VAR word : buffer;
                                        VAR value: INTEGER);
(*Converts hex character string to decimal value:
  buffer is a globally declared type --> PACKED ARRAY[1..8] OF CHAR;
  word contains the hex character string
  first and last are pointers into the string
  value is the returned decimal value                              *)

VAR  i,n,factor      : INTEGER;
     ch              : CHAR;

BEGIN
  value := 0; factor := 1;
  FOR i := last DOWNTO first DO
    BEGIN
    ch := word[i];
    IF ch = ' ' THEN n:=0     (*Blank character given value 0*)
    ELSE
      IF (ch>='0') AND (ch<='9') THEN   (*character range 0..9 *)
        n := ORD(ch)-ORD('0')           (*convert ch to decimal*)
      ELSE
        IF (ch>='A') AND (ch<='F') THEN (*character range A..F *)
          n := ORD(ch) - ORD('A') + 10; (*convert ch to decimal*)
    value := value + factor * n;
    factor := 16 * factor;              (*hex is base 16*)
    END;
END;                                    (*procedure getvalue*)
```

Example function declaration:

```
FUNCTION nextstate(currentstate : INTEGER) : INTEGER;
(* returns the next state given the current state *)

BEGIN
  CASE currentstate OF
    1: nextstate := 3;
    2: nextstate := 4;
    3: nextstate := 1;
    4: nextstate := 2;
  END;
END;                                    (*function nextstate*)
```

B.8 Statement Body

The statement body of a block contains zero or more statements which describe the actions of the block. The statement body must start with the keyword BEGIN and stop with the keyword END. However, since statements may also include BEGIN and END, the statement body may contain many occurrences of these two keywords. The statement bodies for the three types of blocks are identical, except that the concluding END for the program block statement body must be followed by a period while the concluding END for procedure and function statement bodies must be followed by a semicolon.

Syntax of statement body:

Example statement body:

```
BEGIN          (* begin program block statement body *)
  WHILE NOT EOF DO
    BEGIN
    READ(x,y,z);
    x := SQR(x); y := SQR(y); z := SQR(z);
    WRITE('squaredata ' , x , y , z);
    END;
END.           (* end of program *)
```

SIMPLE DATA TYPES

The simple data types are the primitive data types of the language. They form the base for building structured types. The simple data types consist of ordinal types and the REAL type.

A. Ordinal Types

Ordinal types are characterized by a linear ordered set of distinct values which can be mapped on the set of natural numbers. This mapping is actually an enumeration of all the values which the type can take. The predefined ordinal types are INTEGER, CHAR, and BOOLEAN. New ordinal types can be defined by enumerating all the values which the type can take. In addition, new ordinal types may be defined as subranges of other ordinal types.

A.1 The Type INTEGER

Variables declared as type INTEGER may take on values in the range -32768 to +32767. All the arithmetic and relational operators can be used with integer constants and variables. However, the relational operator IN is used only in conjunction with sets (see chapter 4).

Note: Integer calculations which cause an overflow will not generate an overflow error. (eg. MAXINT + 1 = -32768)

Syntax of type INTEGER:

Example declaration:

```
VAR          i,j,k : INTEGER;
```

Example integer constants:

```
59    -1    0     329     -10000    29872
```

A.2 The Type CHAR

Variables declared as type CHAR can take single characters as values. The set of valid single characters is defined by a character set. All characters have an associated ordinal number in the range 0 to 255. A table of ASCII characters with associated ordinal numbers is listed in the appendix. There are two functions which may be used in conjunction with the character set. The function ORD(character) returns the ordinal number of the character specified. The function CHR(ordinal number) returns the character associated with the specified ordinal number. These are known as transfer functions because they are used to transfer a character value to an integer value and vice versa. Constants of type CHAR are denoted by using single character strings. All relational operators may be used with variables and constants of type CHAR.

Syntax of type CHAR:

--> CHAR -->

Example declaration:

```
VAR        alpha , beta    : CHAR;
```

Example character constants:

'9' 'a' '#9F'

Example relational expression:

'A' < 'B'

A.3 The type BOOLEAN

The boolean type represents logical data. A logical value is represented by the predefined identifiers FALSE and TRUE. These are the only possible values of a boolean variable or expression.

Syntax of type BOOLEAN:

--> BOOLEAN -->

The boolean type is defined by the following enumeration:

BOOLEAN = (FALSE, TRUE)

The boolean operators AND, OR, and NOT take boolean operands and yield boolean results. The relational operators = , <> , <= , < , > , >= , and IN all yield boolean results. See chapter 7 for examples of boolean expressions.

Example declaration:

```
VAR  switch   : BOOLEAN;
```

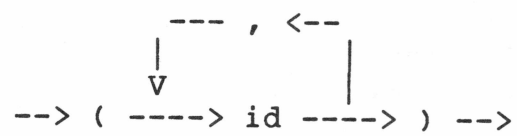
Boolean Constants:

FALSE TRUE

A.4 The Enumerated Type

Pascal allows you to define your own ordinal types. A new type may be created by enumerating all the values that the type may take. This is done by giving the new type a name and listing the values which the new type can take.

Syntax of the enumerated type:



Example definitions of enumerated types:

```
names = (Fred, Joe, Nancy, Susan);

foods = (hotdog, hamburger);
```

The values listed are identifiers. The order in which the identifiers are listed defines a relationship. The identifiers can be thought of as being mapped on to a set of natural numbers. The first identifier maps to 0, the second to one, the third to two, and so on. This implies that identifier1 < identifier2 < identifier3...< identifierN. For example, consider the predefined type:

```
BOOLEAN = (FALSE,TRUE)
```

The boolean value FALSE is less than the boolean value TRUE because FALSE appears in the list before TRUE. This kind of ordered relationship applies to any enumerated type. Consider the type definition:

```
colors = (red, blue, green)
```

By this definition, a variable declared as type colors can take on the the value red, blue, or green. The definition also implies that red < blue < green.

The ordering means that enumerated values can be used in relational expressions. It also means that they may be used for range specifications. For example, consider the FOR statement. The range of the loop control variable is defined by specifying a starting and stopping value. These starting and stopping values could be the values of an enumerated type. For example, if color has been declared as type colors, the following statement is valid:

```
FOR color := red to green DO .....
```

A.5 Subrange Types

A subrange type is simply a type defined to take on a subset of the values representing some ordinal type.

Syntax of the subrange type:

The use of subranges can sometimes save memory. For example, an integer variable whose values are always in the range of 0 to 255 could be declared as a subrange of the type INTEGER. You might define a new type as follows:

```
byte  = 0..255
```

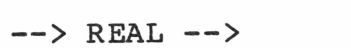
Now, variables declared as type byte would be allocated 8 bits of storage rather than the 16 bits which is allocated for variables declared as type INTEGER. The compiler allocates the minimum amount of storage required to represent the range of values specified by a subrange type.

The use of subranges can better document a program by defining the range of valid values a variable declared as the subrange type can take on. Subrange types are also often used in conjunction with SET types which are discussed in section B of chapter 4.

B. The Type REAL

The type REAL is used to represent fractional numerical data. The implementation of reals is machine dependent. Information on the size, range, and accuracy of reals is discussed in the System Implementation Manual. See section B of chapter 1 for the syntax of real constants.

Syntax of REAL:



Example declaration:

```
VAR       x , y , z    : REAL;
```

Example real constants:

```
2.3      -129.345        5.496E-14       -7983.851D+23
```

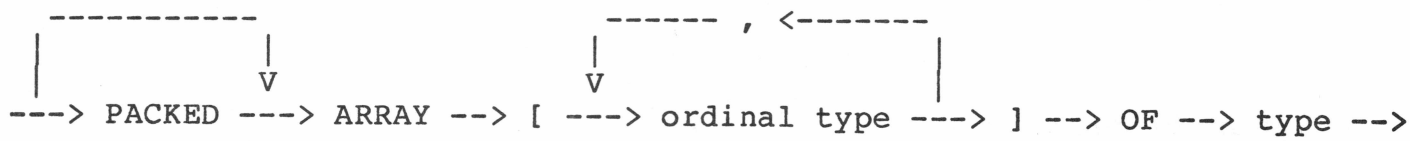
STRUCTURED DATA TYPES

There are four kinds of structured data types: the ARRAY, the SET, the FILE, and the RECORD. These four kinds of data types represent four different ways of organizing the simple data types into a data structure. A data structure can also include other data structures as components. It is then possible to build very complex structures from the basic simple data types. All structured data types can be packed. This means that the most compact form of storage possible will be used. Packing a data structure can sometimes save memory. However, packing may cause access time to increase. The decision to pack or not depends on the specific application. The keyword PACKED signals the compiler to pack the data type into its most compact form. When structured types contain other structured types, the keyword PACKED must be applied to the innermost structure as well as the outermost to have any effect.

A. The Type ARRAY

The ARRAY is a data type which defines a structure composed of a fixed number of data elements which are all of the same type. The data elements can be defined to be of any one type. They could be defined as one of the simple types or as one of the structured types, including ARRAY. Arrays can be defined to be of any dimension. The number of dimensions, the number of elements in each dimension, and how the elements are accessed is specified by an index definition. The index definition consists of a list of ordinal types(excluding the type INTEGER for the reason that this would create an array too large to fit in memory). The number of types specified corresponds to the number of dimensions of the array.

Syntax of the type ARRAY:



Example declarations:

```
TYPE    table = ARRAY [0..5,1..10] OF INTEGER;
        colors = (red, blue, green, yellow);
VAR     report : ARRAY [1..20] OF table;
        day    : ARRAY [1..365] OF REAL;
        class  : ARRAY [0..8,0..5] OF INTEGER;
        chart  : ARRAY [colors] OF INTEGER;
```

Elements of variables declared as type ARRAY are accessed by specifiying the variable name and listing expressions which evaluate to ordinal values that fall into the range of the ordinal types of the index definition.

Examples of accessing array elements:

```
report[5,3,6]      day[40]      class[0,0]      chart[red]
```

B. The Type SET

A set is a collection of distinct elements which are all of the same ordinal type. The elements of a set are called set members. There may be up to 256 members in a set. The 256 member limit causes the restriction that a set can not be defined to be of ordinal type INTEGER. Also, subranges of type INTEGER which include negative integers are not allowed as set base types. A set can have no members in which case it is called an empty set.

Syntax of the type SET:

```
--> SET --> OF --> ordinal type -->
```

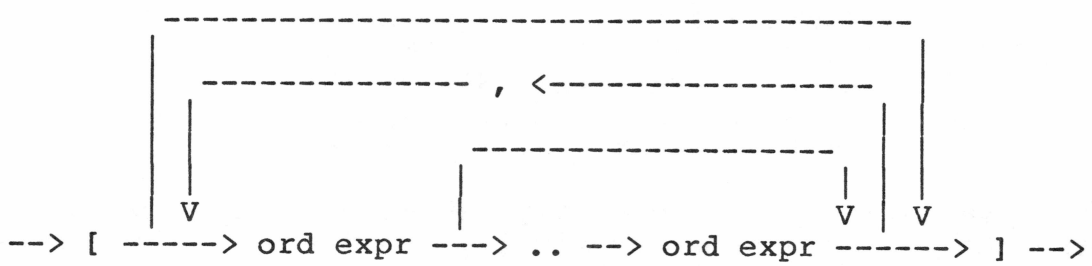
Example declarations:

```
TYPE    days = (sunday, monday, tuesday, wednesday,
                thursday, friday, saturday);

VAR     lowercase, digits, special : SET OF CHAR;
        schooldays, workdays       : SET OF days;
        day                        : days;
```

A variable declared as type SET can take on any values which are subsets (including the empty set) of the values defined by the type of the set. The type of the set is specified after the keyword OF.

Set values are denoted by listing set members within square brackets. The individual members can be specified as ordinal expressions.

Syntax of set notation:



The .. notation between two members specifies that all values in between are also to be included as members. For example, [0..3,7..10] would denote a set with members 0,1,2,3,7,8,9,10. The empty set is denoted by [].

Example assignments to set variables:

```
schooldays:= [monday, wednesday, friday];
workdays  := [monday..friday];
lowercase := ['a'..'z'];
digits    := ['0'..'9']
special   := ['*' , '%' , '@']
```

The relational operators which are applicable to sets are (IN , = , <> , <= , and >=)

IN

A single element can be tested to see if it is a member of a set. The operator IN is used for this testing of set membership. This operation evaluates to TRUE if the single element on the left is a member of the set on the right.

=

Two sets can be compared to see if they contain exactly the same members. The operator = is used to test for set equality. If each member of each set is also a member of the other then the operation evaluates to TRUE.

<> Two sets can be compared to see if they do not contain exactly the same members. The operator <> is used to test for set inequality. If any member of either set is not also a member of the other then the operation evaluates to TRUE.

<=

A set can be compared to another set to see if the first set is a subset of the second set. The operator <= is used to test for set inclusion. If all the members of the set on the left are also members of the set on the right then the operation evaluates to TRUE.

>=

A set can be compared to another set to see if the first set is a superset of the second set. The operator >= is used to test for set containment. If there are no members in the set on the right which are not also members of the set on the left then the operation evaluates to TRUE.

Example use of relational operators:

```
IF day IN workdays THEN gotowork; (*gotowork is a procedure*)
IF character IN digit THEN WRITE(character);
IF workdays >= schooldays THEN noweekendclasses;
```

Relational Expression	Evaluation
monday IN [monday, tuesday]	TRUE
'A' IN ['a'..'z']	FALSE
[1, 2, 3] >= [0]	FALSE
[1, 2, 3] >= [2]	TRUE
['%'] <= ['*', '%']	TRUE
[] <= [tuesday]	TRUE
['a', 'f', 'g'] = ['a', 'f', 'k']	FALSE
[1] = [1]	TRUE

The arithmetic operators which are applicable to sets are (+ , - , and *).

+

Two sets can be combined to form a third set containing all elements that are members of either set. The operator + performs the union of two sets.

-

A set can be formed as the difference between two sets. The operator - performs set difference. The result is a set containing all members of the set on the left which are not also members of the set on the right.

*

A set can be formed which contains only the members which exist in both of two other sets. The operator * performs the intersection of two sets.

Examples:

Expression	Result
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
[1, 2, 3] + [2, 3, 4]	[1, 2, 3, 4]
[1, 2, 3] - [2]	[1, 3]
[1, 2, 3] - [4]	[1, 2, 3]
[1, 2, 3] * [4, 5, 6]	[]
[1, 2, 3] * [2, 3, 4]	[2, 3]

C. The Type FILE

The data type FILE provides the link between a program and the peripheral equipment of the computer system. Variables declared as type FILE represent logical files. Input and output operations always refer to logical files. Each logical file has an associated physical file. The physical file is the actual device to which an operation is directed. A physical file is a device such as a terminal, printer, disk file, etc... Since all input and output operations reference logical files rather than physical files, a programs input or output can be redirected simply by associating the logical file with a different physical file. The method of associating logical files to physical files is discussed in the System Implementation Manual.

File data elements can be of any type except FILE or structured types containing a component of type FILE.

Structured variables (eg. array or record variables) may contain components of type FILE. However, the I/O routines (see chapter 10) will accept only simple variable names. For example, file names such as "customer.file1" or "files[2]" are not accepted by the I/O routines. See the appendix for an example of how to work around this restriction.

Syntax of type FILE:

```
--> FILE --> OF --> type -->
```

Input and output can be greatly simplified by declaring variables as files of structured types. For example, a complete record can be read or written to a file of records simply by specifying the file variable name and the record variable name as parameters to an input or output procedure.

Example of file declarations:

```
Type     sales = RECORD
                 salesman : PACKED ARRAY[1..20] OF CHAR;
                 quantitysold : INTEGER;
                 END;
VAR      salesfile : FILE OF sales;
         numbers   : FILE OF INTEGER;
```

The data elements of files declared as above are read and written in binary format. Binary format is the form in which the data is actually stored in memory. No translation of the data is done during the I/O process to a character readable form. The advantage of this type of I/O is speed of data transfer and minimization of disk storage requirements. The disadvantage is that the data is in a non-readable form.

A special type of file is provided for handling character formatted data. In a TEXT file, data is stored as characters. Input and output then involves a translation to and from the internal binary data format.

C.1 The type TEXT

There is a predefined type of file called TEXT. Text files have special characteristics. Unlike other file types, a text file is divided into lines. There is some mechanism which is implementation dependent which marks the separation between lines, each line being a sequence of characters. The data types which can be input from and output to text files are not restricted to characters only, even though a text file is actually a file of characters. The characters of a text file may represent string, integer, real, or boolean values. The Pascal I/O routines make the appropriate character to binary and binary to character conversions with TEXT files. There are two predeclared variables of type TEXT (INPUT and OUTPUT). These are the default parameters for the I/O procedures and functions discussed in chapter 10.

Example declarations:

```
infile, outfile : TEXT;
```

There are two built in procedures and one built in function which apply only to text files.

WRITELN

The procedure WRITELN terminates the current line and positions a file pointer to the next line. If any variables are specified to be output by the WRITELN, they are output first and then the file pointer is advanced to the next line.

READLN

The procedure READLN causes the file pointer to be positioned to the beginning of the next line. If any variables are specified to be input by the READLN, they are input first and then the file pointer is advanced to the next line.

EOLN

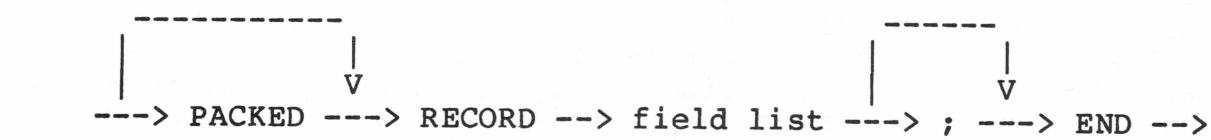
The function EOLN is a boolean function which evaluates to TRUE when the end of a line has been reached. At all other times it evaluates to FALSE.

(See chapter 10 for details)

D. The type RECORD

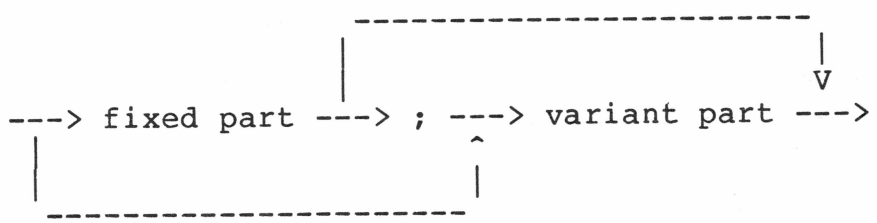
The type RECORD is characterized by a fixed number of elements which are called fields. The fields of a record can be of different types. Record field identifiers can be declared to be of any type, including RECORD. Therefore, records can be nested.

Syntax of the type RECORD:

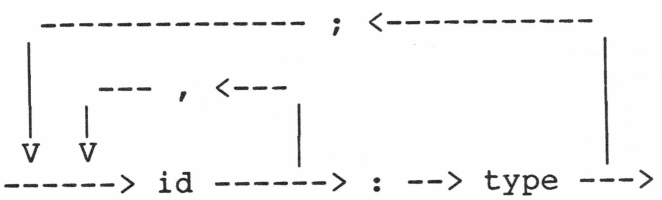


The field list describes the individual components of a record. All the field identifiers within a record must be unique. However, field identifiers are scoped within the record itself which means that an identifier outside the record definition can be identical to a field name within the record. The field list consists of two separate parts, a fixed part and a variant part. A record can contain either or both of these two parts. If both parts are present, the fixed part must precede the variant part. The fixed part refers to the part of the record which is always referenced in the same way (ie. the fields are fixed). The variant part refers to the part of the record which may be referenced in multiple ways (ie. the fields may vary).

Syntax of field list:



Syntax of the fixed part of a record:

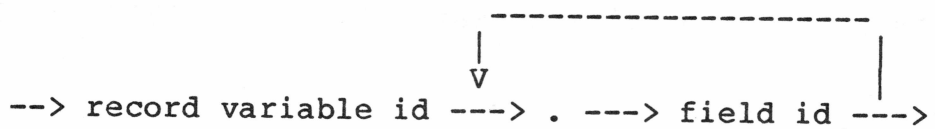


Example record using fixed fields:

```
RECORD
  business: PACKED ARRAY[1..25] OF CHAR;
  location: RECORD
              street,
              city,
              state   : PACKED ARRAY[1..15] OF CHAR;
              zip     : INTEGER;
            END;
END;
```

A particular field of a record variable is referenced by the variable name followed by the field name. A period separates the two names. If the field name is itself a record, then a field within the nested record is referenced by appending a period and the field name to the other two names.

Syntax of record variable referencing:



Example referencing:

Assume that customer is a record variable as defined on the previous page,

then

```
customer.business   references first  field
customer.location   references second field
```

The nested fields of the field "location" are referenced by:

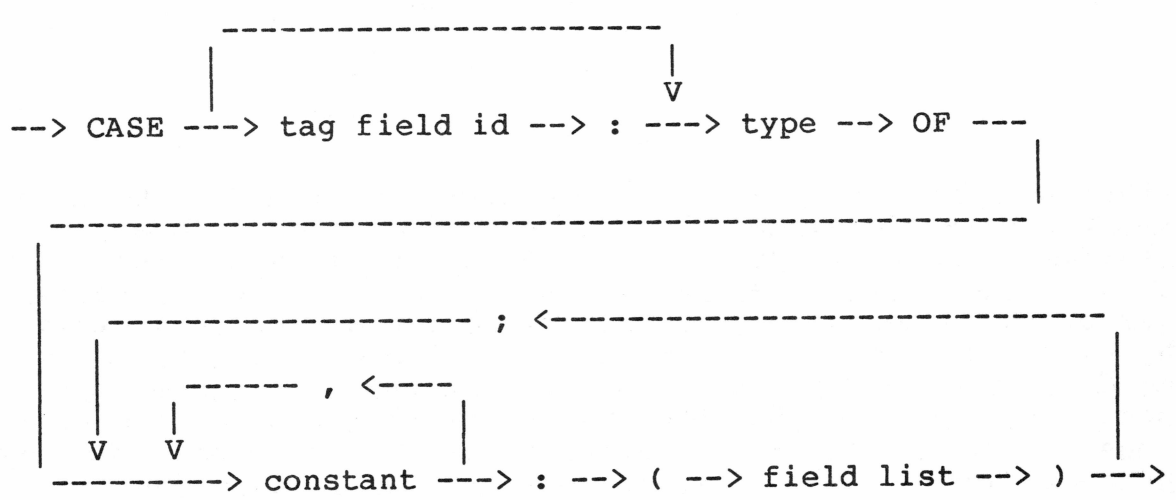
```
customer.location.street
customer.location.city
customer.location.state
customer.location.zip
```

D.1 Record Variants

Sometimes it is useful to be able to define a storage area in a record which can be accessed in multiple ways. Record variants provide the ability to do this. In certain applications, they can simplify a program and save storage space at the same time.

A record variant defines a fixed size storage area of a record which can be accessed in multiple ways. The size is determined by the variant alternative which requires the largest amount of storage space. The variant is defined using a form similar to that of the CASE statement.

Syntax of the variant part of a record:

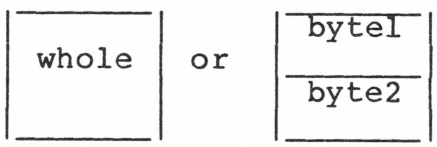


Each alternative way of accessing the storage area of a variant is defined by a field list. All field names within the variant definition must be unique. The storage area can then be accessed in the desired way simply by specifying the appropriate field name. There are two forms of the variant. In one form, a tag is specified and becomes a field in the record. The tag field resides in the record just prior to the variant storage area. The purpose of the tag field is to store a value which specifies for each record the alternative of the variant which is in effect. The other form omits the tag field which in some cases is not needed.

Example using no tag field:

```
PACKED RECORD
  CASE BOOLEAN OF
    FALSE: (whole          :INTEGER);
    TRUE : (byte1,byte2    :0..255;);
END;
```

This variant definition would define a storage area of two bytes (assuming an integer is 16 bits) which is the largest amount of storage required for either of the two field lists. You could then access the whole two byte storage area as an integer or you could access each individual byte of the integer. The storage could be pictured as follows:



The type BOOLEAN was chosen as the selector of the CASE because it defines two possible values which is what is needed to specify the two alternatives. Another type could have been defined and used just as well. With the variant defined as above, you could now reference the integer or the bytes simply by specifying the appropriate field name: whole, bytel, or byte2. For example, if "number" is a variable declared as this record type, then "number.whole", "number.bytel", and "number.byte2" are the possible ways of referencing this storage area. Care must be taken when using variants for this purpose. The way in which the fields of the different forms of the variant overlap one another is implementation dependent. Also in the above example, which byte would be the low byte and which would be the high byte is implementation dependent. (See the System Implementation Manual)

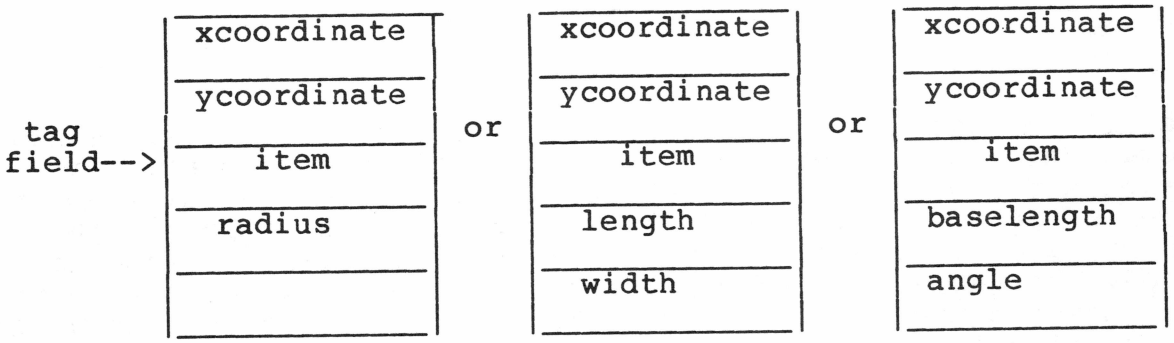
Example using tag field:

Assume the type definition:

```
itemtype = (circle, rectangle, triangle)
```

```
PACKED RECORD
  xcoordinate, ycoordinate :REAL;
  CASE item    :itemtype OF
    circle     :(radius                  :REAL);
    rectangle  :(length,width            :REAL);
    triangle   :(baselength              :REAL;
                 angle                   :INTEGER);
   END;
```

This record definition contains a fixed part as well as a variant part with a tag field. The storage allocation for this record could assume the following structures:



The storage allocated would be the amount required to store the two real numbers of the fixed part, the tag field, and the two real numbers of the rectangle field list. The other field lists of the variant require less storage than the rectangle list. The information of which alternative of the variant is in effect can now be stored as part of each record via the tag field. The tag field is referenced in the same manner as the other fields.

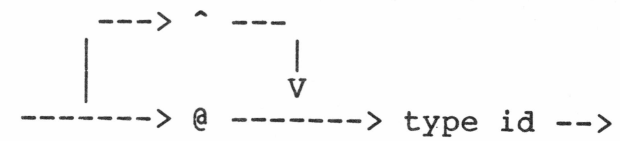
Note:
Variants can be nested. That is, a variant can contain a definition of another variant. However, there can be only one variant at any one level and the variant definition must follow any fixed fields of a record.

POINTER DATA TYPE

The pointer data type is used in conjunction with dynamic storage allocation. This refers to the creation of storage space for variables during program execution. This is very useful when the amount of data storage a program will require is unknown. The use of pointer data types provides the ability to allocate storage as it is needed. Variables for which storage is dynamically created cannot be referenced in the usual manner. The reason is that they actually have no identifiers of their own. Instead, they are referenced through the use of pointers. A pointer is actually a variable which points to the location in memory of a dynamically created variable.

The definition of a pointer type specifies the data type for which storage will be allocated. The data type then determines the amount of storage required for each allocation. The definition of a data type does not have to precede the definition of a pointer type which references it. This is the only exception to the rule that identifiers must be defined before they are used. This allows for a field of a record to be declared as a pointer to the record itself. Either the symbol ^ or the symbol @ may be used to signify a pointer type.

Syntax of type pointer:



Example pointer declarations:

```
TYPE  transptr = @transaction;
      transaction = RECORD
                      item          :INTEGER;
                      price         :REAL;
                      link          :transptr;
                      END;
```

In the above declaration, transptr is a pointer type defined to be a pointer to the data type transaction. Transaction is a record consisting of three components (item, price, and link). Dynamic variables of the type transaction can be created through the use of pointer variables of type transptr. Notice that link is declared to be of type transptr. This component of the record is a pointer variable which may point to another dynamic variable of type transaction. Therefore, a linked list of transaction records can be formed with the link field of each record pointing to the next record.

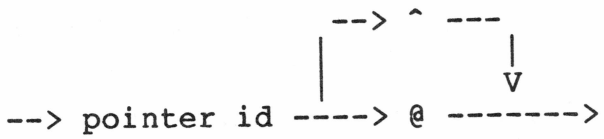
The predeclared procedure NEW is used to allocate storage for dynamic variables. It has one argument which is a pointer variable. The NEW procedure allocates the amount of storage required by the data type associated with the pointer and assigns the address of the allocated storage to the pointer. The pointer is then used to reference the allocated storage. For example, consider the declaration:

```
list : transptr;
```

Then the statement NEW(list) would allocate the amount of space required to store the three components of a transaction record at some location in available memory and assign the location in memory to the variable list. The available memory is called the heap and its size is set at run time. (See the System Implementation Manual)

References to a variable which is pointed to by a pointer are made by following the pointer name with either the symbol ^ or the symbol @. In the above example, list@ would reference the dynamically created transaction record.

Syntax of referencing dynamic variables:



Example referencing of dynamic variables:

list@	references whole record
list@.item list@.price list@.link	references individual fields
list@.link@	references record pointed to by link field
list@.link@.item list@.link@.price list@.link@.link	references individual fields

When a dynamically created variable is no longer needed, it may be disposed of. This is the process of freeing the space consumed by the variable for other uses. The predeclared procedure DISPOSE is provided for this purpose. Like the NEW procedure, it has one parameter which is a pointer. The DISPOSE procedure frees the memory allocated to the variable pointed to by the pointer. Referring to the above example, DISPOSE(list) would free the amount of memory which was allocated to the dynamic transaction variable.

A predefined constant NIL can be used to assign a value to a pointer. Other than using the procedure NEW, assignment to the constant NIL is the only way of giving a pointer a defined value. If a pointers value is NIL, then it does not point to a dynamic variable. This is often used with linked lists to give the pointer of the last element in the list a defined value. It provides a way of detecting when the end of the list has been reached.

Example procedures using pointer variables:

```
PROCEDURE create(VAR translist : transptr);

(* Creates a new transaction
   Adds the transaction to the top of a transaction list
   Returns a pointer to the new transaction via translist
   New transaction becomes top of transaction list*)

VAR
      trans                (*new transaction pointer*)
                  :        transptr;

      (*note: translist should be initialized to NIL*)

BEGIN
  NEW(trans);               (*create new transaction*)
  trans@.link:=translist; (*new transaction points to old top of list*)
  translist:=trans;         (*new transaction becomes top of list*)
END;                        (*procedure create*)
```

```
PROCEDURE   destroy(translist, trans : transptr);

(* Removes the transaction pointed to by trans from the list
   Recovers the memory used by the transaction     *)

VAR
      lead,            (*points to next transaction in list*)
      trail            (*saves location of current transaction
                         while lead is advanced to the next
                         transaction*)
                  :    transptr;

BEGIN
  lead:=translist;
  While lead <> trans DO (*search for trans*)
    BEGIN
    trail:=lead;          (*save pointer to current transaction*)
    lead:=lead@.link;     (*advance pointer to next transaction*)
    END;
  IF translist <> trans THEN (*check if trans is at top of list*)
    trail@.link:=lead@.link  (*link around transaction*)
  ELSE
    translist:=lead@.link;   (*new top of list*)
  DISPOSE(trans);            (*recover memory*)
END;                         (*destroy*)
```

OPERATORS

There are four categories of operators: arithmetic, relational, boolean, and type transfer.

A. Arithmetic Operators

The following table lists all the arithmetic operators, the operations they perform, the type of operands which may be used, and the type of result of the operation. Mixed mode arithmetic is supported. (eg. it is allowed to have an integer value added to a real value) Also, automatic truncation occurs when an integer variable is assigned a real value.

Operator	Operation	Type of Operands	Type of Result
+	addition	integer, real	integer, real
	set union	sets of compatible types	same type as the larger set
-	subtraction	integer, real	integer, real
	set difference	sets of compatible types	same type as the larger set
*	multiplication	integer, real	integer, real
	set intersection	sets of compatible types	same type as the larger set
/	division	integer, real	real
DIV	truncated division	integer	integer
MOD	modulus	integer	integer

Note: For sets to be of compatible types they must have identical base types , one base type must be a subrange of the other, or they may both be subranges of the same base type.

B. Relational Operators

All relational operators perform operations which yield Boolean results. The result is always either TRUE or FALSE. In general, both operands of a relational operator must be expressions of identical type, but the types REAL, INTEGER, and subranges of integer may be mixed.

(Relational operations may be performed on any types except files)

Operator	Result of Operation
=	true if left operand is equal to right
<>	true if left operand is not equal to right
<	true if left operand is less than right
>	true if left operand is greater than right
<=	true if left operand is less than or equal to right
>=	true if left operand is greater than or equal to right

To compare strings, the ordinal numbers of the characters composing both strings are compared to one another until a pair of characters are different or until the end of the strings is reached. If there are no character pairs which differ then the strings are equal. Otherwise, the first pair of characters which differ determine the relationship. The string whose character ordinal number is the largest is greater than the other string.

Operation	Result
'abc' = 'cdf'	FALSE
'abc' < 'abd'	TRUE
'bab' > 'adf'	TRUE

The following operator tests for set membership. The left operand may be any ordinal type and the right operand may be any set of the same ordinal type.

IN	true if left operand is a member of the right (See section B of chapter 4)

C. Boolean Operators

The boolean operators, like the relational operators yield boolean results. The result is always either TRUE or FALSE. The operands of a boolean operator must be boolean expressions.

Operator	Result of Operation
OR	true if either one or both of the operands is true
AND	true only if both operands are true
NOT	true if operand is false

Operation	Result
FALSE OR FALSE	FALSE
TRUE OR FALSE	TRUE
FALSE OR TRUE	TRUE
TRUE OR TRUE	TRUE
FALSE AND FALSE	FALSE
TRUE AND FALSE	FALSE
FALSE AND TRUE	FALSE
TRUE AND TRUE	TRUE
NOT TRUE	FALSE
NOT FALSE	TRUE

D. Operator Precedence

Operator precedence defines the order in which operations take place within expressions. In general, expressions are evaluated from left to right. However, operations of higher precedence are performed before operations of lower precedence. All operators are ranked by precedence. Parentheses have the highest precedence and may be used to alter the normal order of evaluation. Nested parentheses are evaluated from the inside out.

Following is a list of the operators arranged by precedence. Operators listed on the same line have equal precedence. The precedence has been slightly altered from the Jensen & Wirth standard to eliminate excessive use of parentheses. The operators NOT, AND, and OR have been altered. In the standard, NOT immediately precedes the unary operators, AND is the same precedence as * etc., and OR is the same precedence as + , -. This alteration should not effect porting standard Pascal to TRS-80 Pascal. However, if porting from TRS-80 Pascal to some other Pascal, you should parenthesize expressions just as you would had the precedence not been altered.

```
  Highest
Precedence--> ()

              + , -  when used as unary operators

              * , / , DIV , MOD

              + , -

              = , <> , < , > , <= , >= , IN

              NOT

              AND
  Lowest
Precedence--> OR
```

Operation	Equivalent To	Result
8+3*4	8+(3*4)	20
10-8/4*2	10-((8/4)*2)	6
5 MOD 10-5	(5 MOD 10)-5	0
3<2 OR 6>8 AND TRUE	(3<2) OR ((6>8) AND (TRUE))	FALSE
NOT 7*2<5	NOT ((7*2)<5)	TRUE

E. Type Transfer

The type transfer operator is used to temporarily change the type of an existing variable. This is useful when there is a need to reference a variable in a manner which would normally not be allowed by Pascal. For example, you might wish to access the lower and upper bytes of an integer variable. The type transfer operator allows you to access parts of variables. Also, it provides a mechanism for avoiding compiler type checking. This may be used in some cases where parameters of differing types must be passed to a procedure.

Syntax of type transfer:

A type transferred variable may be used wherever a variable is allowed. Regardless of its original type, the type transferred variable is then accessed according to the type indicated. The type transfer operator tells the compiler to treat the variable as if it were of the new type. No data conversion takes place. The variable is simply referenced as if it were of the new type. Type transferred variables must adhere to the same type matching rules as normal variables.

Example use of type transfer operator:

```
TYPE        byte = 0..#FF;
            integrec = PACKED RECORD
                       upper, lower     :byte;
                       END;
            pointer = @integrec;

VAR         number : ARRAY[1..10] OF INTEGER;
            integr : integrec;
            address: pointer;
```

Valid type transfer operations:

```
integr.upper := number[1]::byte;
number[1]::byte := integr.lower;
READ(integr::INTEGER);
number[5] := address::INTEGER;
address::INTEGER := 25 + number[3];
```

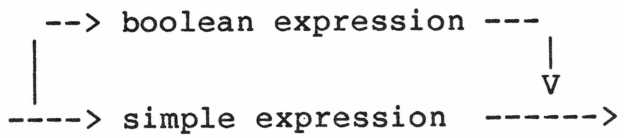
The fundamental use of type transfer is to overlay a type template on a data structure so that components of the structure may be treated as if they were of any desired type. This requires a precise understanding of how the compiler represents the data type (how it is stored) in order to insure the operation does what was intended. Because of this, it should be used with caution and only when necessary. (See the System Implementation Manual)

EXPRESSIONS

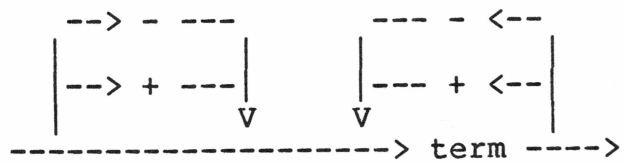
An expression is a variable, a constant, a function call, a set notation, or a combination of these operands with a description of the operations to be performed on them. The operators and operands of an expression define an implicit type for the expression. When evaluated, the expression yields a value of that type. For example, an integer expression is composed of operands and operators which when evaluated yield an integer result, a real expression yields a value of the type REAL, an ordinal expression yields a value which is of one of the ordinal data types, etc...

An expression can be just a simple expression or it can be a boolean expression. A simple expression can yield a value of any data type. A boolean expression is composed of simple expressions but always yields a value of the type BOOLEAN.

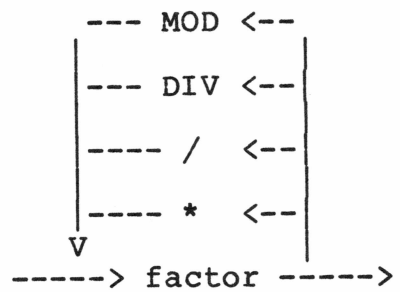
Syntax of expression:



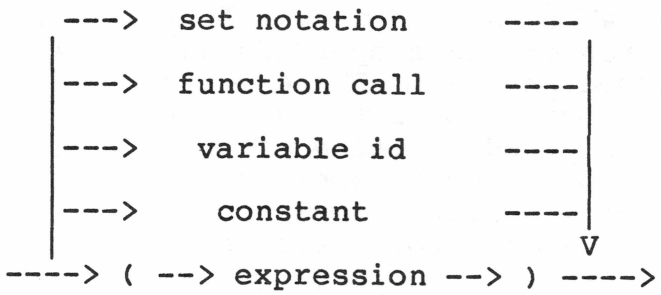
Syntax of simple expression:



Syntax of term:



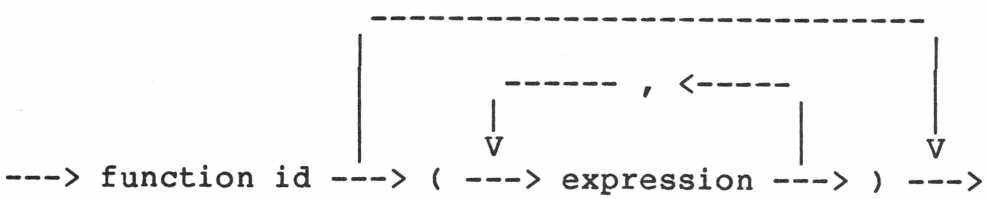
Syntax of factor:



For the syntax of set notation, refer to the structured data type SET. A function call has the same form as the procedure statement. The only difference is that a procedure call is a statement while a function call is a part of an expression. Remember that a function has a type associated with it. When a function call is encountered in an expression, the named function is activated. Somewhere in the function a value is assigned to the function name. When the function terminates, the value assigned to it is substituted in the expression for the function call.

Note : a function may be an ordinal type or the type REAL only.

Syntax of a function call:



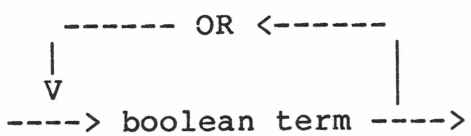
Example function calls:

```
salary

payment(interestrate,years)

sum(a+b)
```

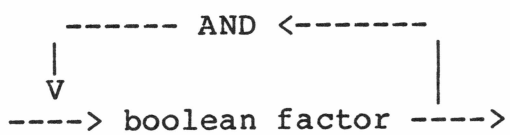
Example simple expressions:

Expression	Result
time	same type as time
weekday + [saturday,sunday]	set
12*payment(interestrate,years)	integer or real depending on type of function "payment"
entry MOD size	integer
-10 DIV 4 + 9.2/6 -45	real
(var1+var2)*153/(var3-var4)	real

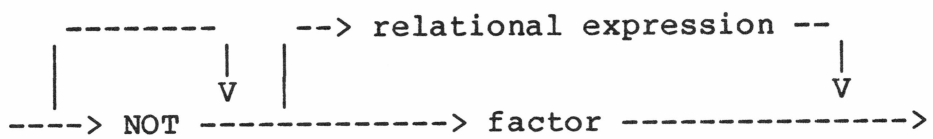
Syntax of boolean expression:



Syntax of boolean term:

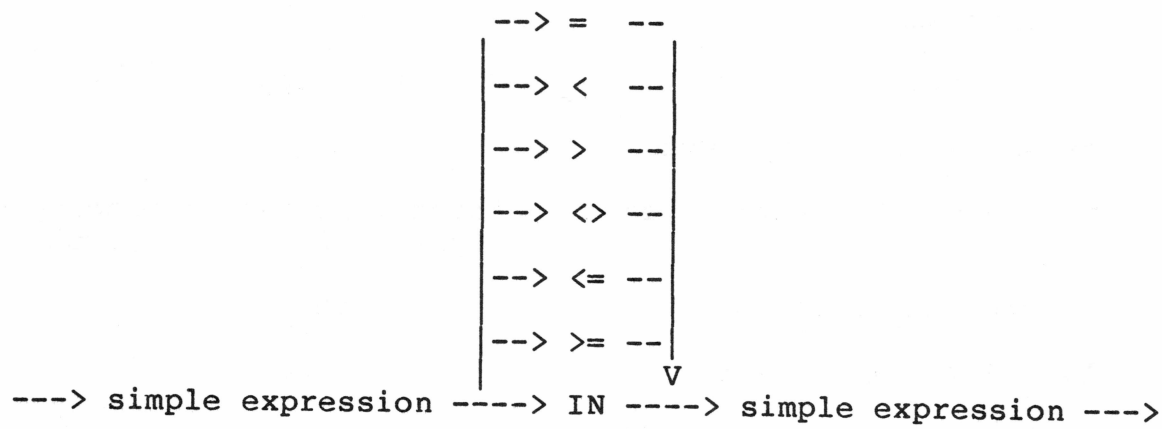


Syntax of boolean factor:



note: factor must be of type BOOLEAN

Syntax of relational expression:



Example boolean expressions:

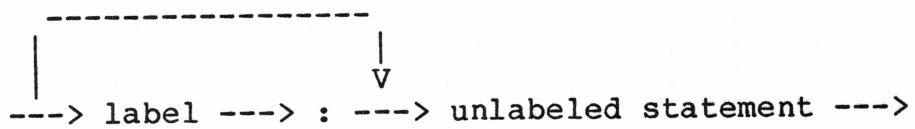
```
a=b OR c<d AND switch

n1 + n2 >= 20 AND n3-n4 <= 11

NOT here OR there

NOT alpha < beta AND gamma <> 'R'

number IN [1..15] OR NOT letter IN ['a'..'z']
```

STATEMENTS

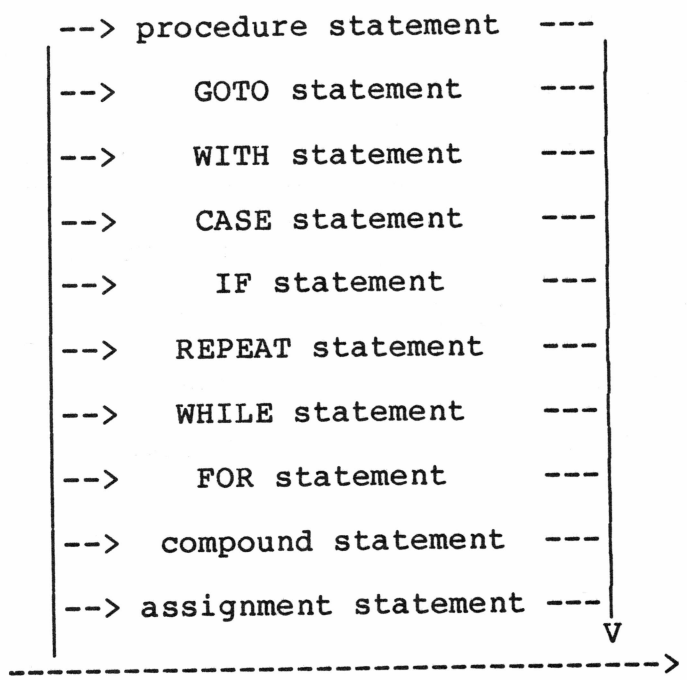
Statements are the Pascal sentences that describe the actions and logic of a program. Statements reside in the statement body part of a block.

A statement may be labeled or unlabeled. A labeled statement is used in conjunction with the GOTO statement. If a statement is labeled, the label must be declared in the LABEL declaration part of the block in which the statement appears.

Syntax of a statement:



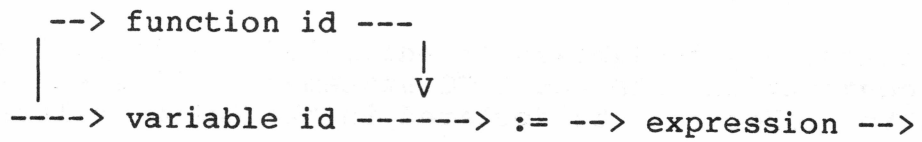
Syntax of an unlabeled statement:



A. The Assignment Statement

The assignment statement is used to assign values to variables and function identifiers.

Syntax of the assignment statement:



The action of the assignment statement is to give the variable or function identifier on the left side of the equal sign, the value of the evaluated expression on the right side. The variable may be of any type. In general, the type of the variable or function must be the same as the type of the evaluated expression. However, there are some exceptions. An identifier of type REAL may be assigned a value which is an integer or a subrange thereof. One side may be a subrange of the other but the value to be assigned should be in the range of the left side. If the identifier on the left side is a SET type, it may be assigned to a set which differs in type as long as the set members of the right side are allowable members of the set on the left side.

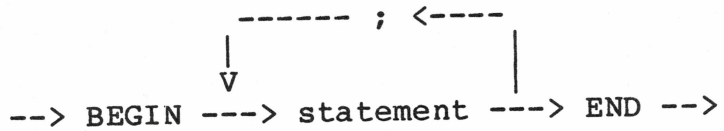
Example assignment statements:

Assignment	left hand side identifier types
a := 10	integer or real
x := 100.5 + 49 + 87/12	real
y := abs(10*z-30.3)	real
test := sample < 10	boolean

B. The Compound Statement

Statements which are bracketed by the two keywords BEGIN and END make up what is termed a compound statement. The compound statement is used in places where more than one statement is required. The compound statement is essential for most of the control structures of Pascal. For example, the FOR statement is a control structure used for executing a statement repeatedly for a specified number of times. The compound statement provides the ability to use this construct for executing a sequence of statements rather than just one.

Syntax of the compound statement:



Example compound statement:

```
BEGIN
  a := b * c;
  d := a/10 + 16.9;
  e := d - 28.3 + 14;
END
```

C. Repetitve Statements

Repetitive statements are the structures used for loop control. They specify that a statement or sequence of statements is to be executed repeatedly until some terminating condition occurs. Pascal provides three such control structures.

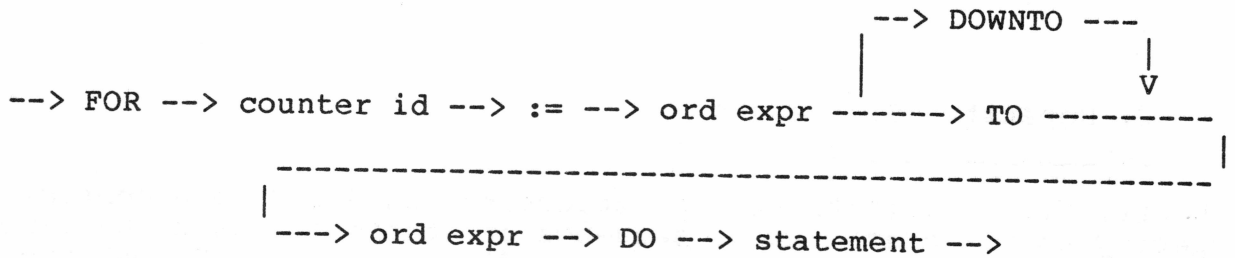
C.1 The FOR Statement

The FOR loop is used when a statement is to be executed a predefined number of times. The FOR loop is characterized by a loop variable which serves as a counter for controlling the number of times a statement is executed. The counter has defined starting and ending values which are ordinal expressions. The expressions are evaluated once upon entry into the loop. At the beginning of each time through the loop, the counters value is compared to the ending value to determine whether or not to end execution of the FOR. At the end of each time through the loop the counters value changes by 1. If the keyword TO is used, the counter is incremented each time through the loop, while the use of the keyword DOWNTO causes the counter to be decremented. The loop is terminated when the counter has incremented or decremented past the ending value. The FOR statement is not executed if the counters starting value is such that the ending value would never be reached. For example, if the starting value was -1, the ending value was 2, and DOWNTO was used, the FOR statement would not be executed.

Note:

The compiler option FORDECL may be used to cause the compiler to generate temporary variables for FOR loop counters. When this option is used, it is not necessary to declare the counter variable.

Syntax of the FOR statement: (counter must be ordinal type)



Example FOR statements:

```
FOR i := 1 TO 30 DO WRITELN(' this gets written 30 times')
---------
FOR j := first DOWNTO last DO
  BEGIN
  initialscore[j] := 0;
  time[j] := 60;
  END
```

C.2 The WHILE Statement

The WHILE statement uses a boolean expression to control repeated execution of a statement.

Syntax of the WHILE statement:

```
--> WHILE --> boolean expr --> DO --> statement -->
```

The evaluation of the boolean expression precedes the execution of the statement. If the expression evaluates to TRUE, the statement is executed and then the expression is reevaluated. This loop continues until the expression evaluates to FALSE. The first occurrence of a FALSE evaluation causes termination of the WHILE statement.

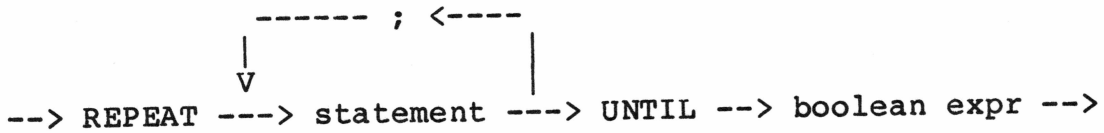
Example WHILE statements:

```
WHILE NOT EOLN DO READ(character)

--------

WHILE (a<b) AND (b<c) DO
     BEGIN
     WRITELN(a,b,c);
     a := a + 1;
     c := c - 1;
     END
```

C.3 The REPEAT Statement

The REPEAT statement, like the WHILE, uses a boolean expression to control repeated execution.

Syntax of the REPEAT statement:



The REPEAT statement is defined such that a sequence of statements which are bracketed by the two keywords REPEAT and UNTIL will be executed at least once. Following the keyword UNTIL is a boolean expression. If the expression evaluates to FALSE then execution returns to the first statement following the REPEAT keyword. If the expression evaluates to TRUE then execution continues with the statement following the boolean expression.

Example REPEAT statement:

```
REPEAT
  i    := i+1;
  j    := j-1;
  k[j] := (i + j) MOD 100;
  l[i] := (i + j) MOD 200;
UNTIL i=j
```

D. Conditional Statements

Conditional statements are used when the execution of a statement must be controlled by some predetermined condition or when one statement out of a group of statements is to be selected for execution. There are two conditional statements.

D.1 The IF Statement

The IF statement uses a boolean expression to control the execution of statements.

Syntax of the IF statement:

In its simplest form, the IF statement involves the evaluation of a boolean expression to determine whether or not to execute an associated statement which follows the keyword THEN. If the expression is TRUE, then the statement is executed, otherwise it is not. The IF statement can also contain an ELSE clause. In this form, if the boolean expression is TRUE, then the statement following the keyword THEN is executed, otherwise the statement following the keyword ELSE is executed.

Example IF statements:

```
IF finished THEN WRITELN(' operation complete');


IF number < 10 THEN range := 1 ELSE range :=2;


IF alpha >= '0' AND alpha <= '9' THEN digit(alpha)
ELSE
  IF alpha >= 'A' AND alpha <= 'Z' THEN letter(alpha)
  ELSE
    special(alpha);


IF contextlist = NIL THEN
  BEGIN
  NEW(context);
  context@.link := NIL;
  contextlist := context;
  END
ELSE
  BEGIN
  temp := context;
  NEW(context);
  temp@.link := context;
  context@.link := NIL;
  END;
```

The statements following the keywords THEN or ELSE can themselves be IF statements. In some forms, an ambiguity can exist in determining which ELSE clause goes with which IF. For example, consider the following case where b1 and b2 represent boolean expressions and s1 and s2 represent statements.

```
IF b1 THEN IF b2 THEN s1 ELSE s2
```

The ELSE could go with the first IF or the second IF. The rule used for solving the ambiguity is to associate an ELSE clause with the nearest IF. The above statement would then be equivalent to:

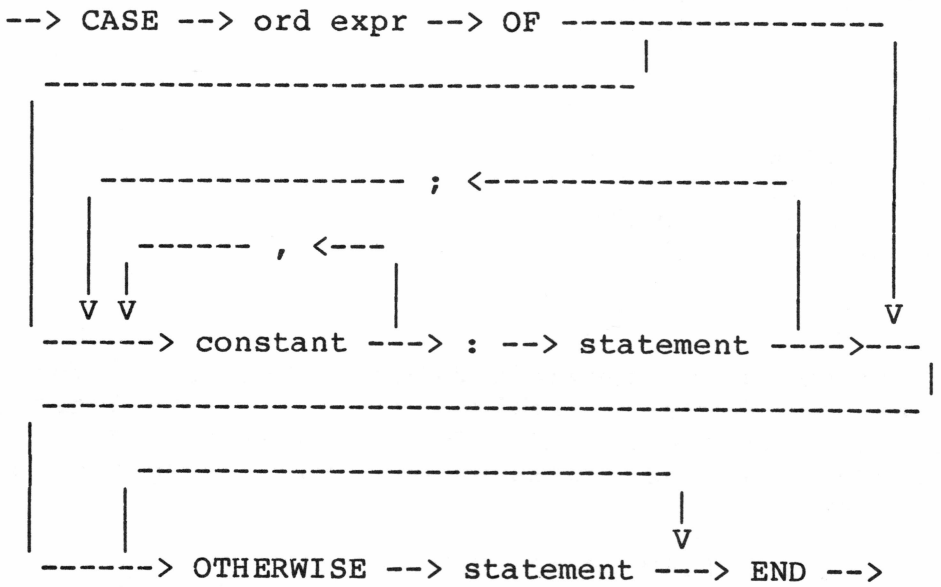
```
IF b1 THEN
     BEGIN
     IF b2 THEN s1 ELSE s2
     END
```

Caution: Semicolons must not appear in the middle of a statement. The most common error for beginning programmers is to put a semicolon in an IF statement which has an ELSE clause. While semicolons are necessary for separation of the individual statements within a compound statement, they must not separate an ELSE from its corresponding IF.

D.2 The CASE Statement

The CASE statement uses an ordinal expression to select one statement out of a group of statements for execution. The group of statements represent alternatives. When a CASE statement is executed, one of the alternatives is selected and executed and then control passes to the statement following the CASE statement.

Syntax of the CASE statement:



The alternative statements of a CASE statement are preceded by constants. The ordinal expression is evaluated and compared to the constants preceding the alternative statements. If a match is found, the statement which has the preceding constant that matches the evaluated expression is executed. There are two actions which can take place in the event that no match is found. By using the OTHERWISE clause, you may specify a statement to be executed when no match is found. If the OTHERWISE clause is omitted and no match is found, then execution continues with the statement which follows the CASE statement.

Example CASE statements:

```
CASE n1+n2 OF
  10: x := sin(x);
  11: x := cos(x);
  12: x := ln(x);
END;
```

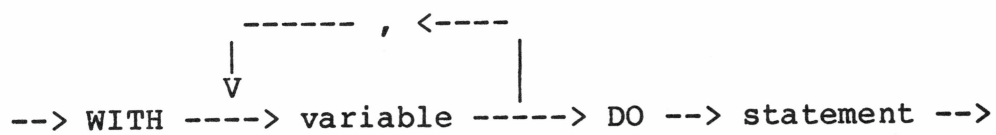
```
CASE ch OF
  'a','b','c': token := 0;
  'd','e','f': token := 1;
  OTHERWISE    token := 2;
END;
```

```
CASE day OF
  monday      : snack := apple;
  tuesday     : snack := orange;
  wednesday   : snack := grapes;
  thursday    : snack := pear;
  friday      : snack := candy;
  saturday,
  sunday      : BEGIN
                weekend := TRUE;
                snack   := nothing;
                END;
END;
```

E. The WITH Statement

The WITH statement is used in conjunction with variables of type RECORD. It makes it possible to use a shorter notation when referencing fields of record variables.

Syntax of the WITH statement:



The variable list specifies the record variables whose fields are to be referenced simply by specifiying the field name itself. When fields of a record are nested (ie. a record is defined as a field of another record), the record variable and the fields, down to the level of the field which is to be referenced in short notation, may be specified in the variable list. Then the nested field can be referenced in the statement simply by specifying its field name. There is a conflict inside the WITH statement when an identifier corresponds to both a variable name and a field name of one of the specified records. For example, you could have a record variable named "weekday" with a field named "monday" and also a simple variable named "monday". Then the following WITH statement might be used.

```
WITH weekday DO monday := 1
```

In such a case, the field name takes precedence over the variable name and the field of the record is referenced. If nested WITH statements are used and a field name inside occurs in more than one of the specified records, then the closest WITH takes precedence.

Example WITH statements:

Assume the declarations:

```
customer : RECORD
           name,
           address,
           city     :PACKED ARRAY[1..20] OF CHAR;
           date     :RECORD
                     month,
                     day,
                     year          :INTEGER;
                     END;
           END;
---------------

WITH customer DO
  BEGIN
  name    :=  'JACK SLATE           ';
  address :=  '1216 MELODY LANE     ';
  city    :=  'TULSA, OKLAHOMA      ';
  END;

WITH customer.date DO
  BEGIN
  month   := 10;
  day     := 23;
  year    := 1981;
  END;
```

F. The GOTO Statement

The GOTO statement is used to cause an unconditional branch to a labeled statement.

Syntax of the GOTO statement:

```
--> GOTO --> label -->
```

The label must be declared in the LABEL declaration part of the same block which contains the GOTO referencing it. The GOTO statement cannot specify a branch to a label outside the block in which it resides. Care must be taken when using the GOTO statement. For example, you should not branch inside a FOR loop from a statement outside the loop. This could cause some very unpredictable results.

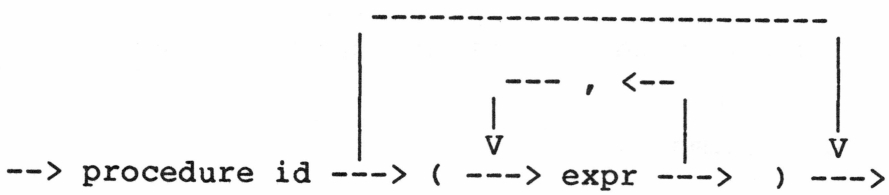
Example GOTO statement:

```
      FOR i := 1 TO 1000 DO
        IF a(i) <> b(i) THEN GOTO 10
        ELSE a(i) := b(i);
10: a(i) := '#0D';
```

G. The Procedure Statement

The procedure statement causes the activation of a procedure. Control passes to the named procedure and then returns to the statement following the procedure statement when the activated procedure terminates. If a procedure has a parameter list, a procedure statement which activates it must specify an argument for each parameter of the parameter list. The arguments must match the order and type of the parameters specified in the parameter list of the procedure. An argument is specified as an expression. If a parameter of a procedure is a pass by reference parameter (denoted by VAR), the corresponding argument of a procedure statement must be a single variable name. The variable may be a simple variable or a component of a structured variable.

Syntax of a procedure statement:



Example procedure statement (call):

(See the procedure declaration in section B.7 of chapter 2)

```
getvalue(n+j,8,hexstring,value)
report
writeout(x,y,3.7+9.6/z)
```

PROCEDURES AND FUNCTIONS

(See chapter 2 for a description of the syntax of procedure and function declarations. A discussion of parameter passing is included with the discussion of the procedure heading.)

Procedures and functions are the tools used to modularize a program. This is the process of breaking a program up into smaller and more manageable pieces. They make a program much more readable and make possible later modifications much easier to handle.

Procedures and functions can be compiled separately and then linked to programs that use them. This allows for the development of libraries of commonly used procedures and functions. Then all the programs that use them can link them in rather than having to include them in the program itself.

The variables declared in a procedure or function do not occupy storage space until the procedure or function is activated. When activated, storage space is allocated for the variables and when the procedure or function terminates, the allocated space is released. Therefore, the amount of storage (or stack) space required by a program at any point in time is a function of the number of blocks which are activated at that time.

A procedure is activated (or called) by a procedure statement. When a procedure is called, control is passed from the point of the call to the procedure. The statements in the procedure then are executed. When the block END of the procedure is reached or when a call to the ESCAPE procedure is made, control passes back to the statement following that which activated the procedure.

A function is activated by an expression. When an expression which contains a reference to a function is evaluated, the function reference causes control to pass to the named function. The statements in the function then are executed. Unlike procedures, functions have a declared type. At some point inside the statement body of a function, the function name should be assigned a value. The value must be the same type as the type to which the function is declared. When the block END of the function is reached or when a call to the ESCAPE procedure is made, control passes back to the evaluation of the expression which activated the function and the function reference is replaced by the value assigned to the function.

A. Scope Rules

A procedure or function declaration forms a new block which is a subblock of the block in which the declaration appears. The new block formed is "nested" within the block which declares it. This process of nesting which occurs every time a procedure or function is declared produces a program structure such as the one shown on the first page of chapter 2. Any block which is enclosed by another block is said to be nested within that block. The level numbers on the diagram indicate how deep the nesting goes beyond the program block which is arbitrarily assigned level 1. The existence of procedures and functions makes it necessary to talk about scope rules. Scope rules describe the accessibility of identifiers from any particular place in a program. The two terms local and global are helpful in discussing scope rules.

An identifier is considered to be local to a block if the identifier is declared within the same block. If there are no blocks nested within the declaring block, then a local identifier can only be referenced by the block which declares it. Enclosing blocks cannot access a local identifier.

An identifier is considered to be global to blocks which are nested within the block in which the identifier is declared. If an identifier is global to a particular block, then that block can reference the identifier provided that it has not declared an identifier of the same name. If a block declares an identifier with the same name as a global identifier, then the global identifier is no longer accessible from that block. Also, any further nested blocks will not have access to the original global identifier.

Identifiers declared in the program block are accessible from any place in a program because all other blocks are nested within the program block. Therefore, identifiers declared in the program block are global to all procedures and functions of the program. Identifiers declared in a procedure or function are local to that procedure or function. The only places in the program which can access these identifiers are the procedure or function itself and the procedures or functions, if any, which are nested within. The nested procedures or functions can access only the global identifiers which they do not declare themselves.

A procedure or function declaration consists of a heading followed by a block. It is important to note that the procedure or function name of a heading is local to the block which declares it. The parameters of the heading are local to the procedure or function itself. This means for example that a procedure statement in the program block can reference any procedure declared in the program block. However, a procedure statement in the program block can not reference any procedure declared within one of these procedures.

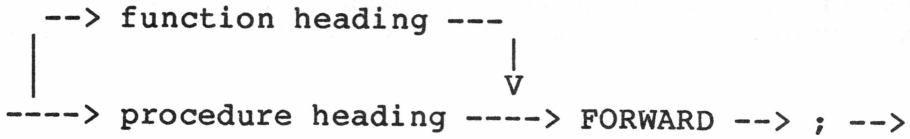
As an example of how scoping effects the accessibility of identifiers, consider the sample diagram on the first page of chapter 2. The following table shows for each block of the diagram, the procedures and functions which are callable from that block, and the constants, types, variables, etc. which can be referenced by the block.

Block	accessible procedures and functions	accessible constants, types, variables, etc.
A	B, D, F	A
B	B, C, D, F	A, B
C	B, C, D, F	A, B, C
D	B, D, E, F	A, D
E	B, D, E, F	A, D, E
F	B, D, F	A, F

B. FORWARD

The rule that an identifier must be declared before it is referenced means that a procedure or function must be declared before it is referenced by a procedure statement or by an expression with a function reference. Some calling sequences that occur among a group of procedures or functions make it impossible to obey this rule. For example, if two procedures call each other, then you can not declare one without referencing the other. The keyword FORWARD provides the mechanism for getting around this problem. Using the keyword FORWARD with just the heading for a procedure or function declaration signals the compiler that the procedure or function block will be declared at some later point in the program. If the procedure or function has parameters, the parameters are declared as well. Then the procedure or function which has been forward declared may be referenced.

Syntax of forward declaring a procedure or function:



(See chapter 2 for the syntax of procedure and function headings)

The actual declaration of a forward declared procedure or function can appear at some later place in the program. The place that it appears must be at the same level and scope as its forward declaration. The actual declaration consists of the heading with no parameters, followed by the block. Since the parameters were declared in the forward declaration, they must not be declared again in the actual declaration.

If a forward declared procedure or function does not have its actual declaration present, then it is treated as an external procedure or function.

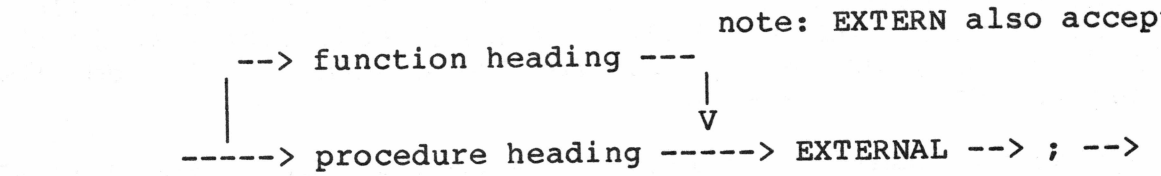
Example use of forward:

```
PROCEDURE abc(pl, p2 : INTEGER); FORWARD;

PROCEDURE xyz;
VAR    pl, p2 : INTEGER;
BEGIN
abc(pl,p2);
END;

PROCEDURE abc;
BEGIN
 .....

END;
```

C. EXTERNAL

An external procedure or function can be declared in a program by specifying its heading followed by the keyword EXTERNAL.

Syntax of externally declared procedures or functions:



Note:
for brevity the word "routine" will be used in place of "procedure or function" in the following discussion.

The linking loader may be used to link separately compiled routines to a program. By declaring a routine to be external, the actual declaration does not have to appear in the program. This is very useful when working with large programs. A large program may be broken up into many routines which are declared as external. The external routines can then be compiled individually. The linking loader can then be used to link the compiled program to its individually compiled routines. One advantage to this is that any changes which are made to a particular routine will cause only that routine to have to be recompiled. The linking process is then repeated after the changed routine has been recompiled. Another advantage is that slightly larger programs can be created by compiling them in pieces and then linking the pieces together.

Perhaps one of most frequent use of external routines is to create a file or library of commonly used routines. Then all the programs which use the routines can link to them rather than having to declare them in each program.

A compiler option must be used to compile a routine by itself. The reason is that a routine by itself is not a legal Pascal program. Therefore, a legal program must be constructed around the routine. This would include a program heading, the environment of the routine, the procedure or function declaration, and a statement body. The environment consists of any constants or types which are in the scope of and are used by the external routine. If global constants or types are needed by the routine, they should be given the same names as those used in the programs that use the routine. The scope refers to the identifiers in a program which are accessible to the externally declared routine.

Variables can also be included in the environment but this is not recommended. If an external routine needs to access a global variable, the variable should be passed as a parameter to the routine. Otherwise, extreme care must be taken to assure that the environment around the external routine matches the environment of the programs which use the routine. The statement body contains the compiler option which is called "nullbody". The nullbody option tells the compiler not to generate any code for the program. Only code for the declared routine is generated.

The syntax for using the nullbody compiler option is shown in the appendix along with all the other compiler options. An example using global variables in an external procedure is also given.

Example use of external procedure:

```
PROGRAM sample;
CONST     .....
TYPE      .....
VAR       xmin,xmax,ymin,ymax : REAL;
          .....

PROCEDURE axes(xmin,xmax,ymin,ymax : REAL); EXTERNAL;
BEGIN
    ....
  axes(xmin,xmax,ymin,ymax);
    ....
END.       (*sample*)
```

Separate compile of procedure axes:

```
PROGRAM axesroutine;
(*global environment, if any, goes here*)
PROCEDURE axes(xmin,xmax,ymin,ymax : REAL);
  TYPE  ....
  VAR   ....
  BEGIN
    .....
    .....
  END;      (*procedure axes*)
BEGIN
  (*$NULLBODY*)
END.
```

D. Recursion

Pascal is a language which supports recursion. Recursion refers to having more than one activation of a particular procedure or function at the same time. There are two forms of recursion. Direct recursion refers to a procedure or function that calls itself. Indirect recursion refers to a procedure or function that makes a call which eventually results in the procedure or function being called again. An example of this is two procedures that call each other. When writing recursive procedures, some conditional statement must exist in the procedure to halt the recursion at some point. Otherwise, there would be an endless loop that would terminate only after the stack was exhausted crashing the program. Recall that each activation of the procedure results in space being allocated for its variables.

Example use of recursion:

```
PROCEDURE XYZ;
  (*DECLARATION HERE*)
BEGIN
.....
   XYZ;  (*PROCEDURE CALLS ITSELF*)
.....
END;
```

E. Predeclared Procedures and Functions

The predeclared procedures and functions are accessible from any place in a program. They are declared in an imaginary block which surrounds the program block. The names of predeclared procedures or functions may be used as identifiers in programs. This means that the name of a predeclared procedure or function may be used in a declaration. If so, then the predeclared procedure or function whose name is used in a declaration is no longer accessible to the program. Its name is associated with the new declaration.

File Associated Procedures

RESET(f)	Positions the file pointer of the specified file to the beginning for the purpose of reading. If the file is empty, then the function EOF becomes true, else it is false.
REWRITE(f)	Replaces the specified file with an empty file. The file pointer is positioned to the beginning of the file.
PAGE(f)	Outputs a formfeed to the specified file. Formfeeds cause skipping to the top of the next page when the file is printed.
CLOSE(f)	Closes the specified file. This procedure may be used to explicitly close a file at any time.
MESSAGE(s)	Outputs the specified string to the terminal. s is char or array of char
READ , READLN WRITE, WRITELN	Read data from a device Write data to a device (See chapter 10 for details)

Arithmetic Functions

	Operation	Type of x	Type of Result
ABS(x)	absolute value	integer, real	same type as x
SQR(x)	square	integer, real	same type as x
SIN(x)	sine	integer, real	real
COS(x)	cosine	integer, real	real
ARCTAN(x)	arctangent	integer, real	real
EXP(x)	natural (base e) exponential	integer, real	real
LN(x)	natural logarithm	integer, real	real
SQRT(x)	square root	integer, real	real

Boolean Functions

ODD(x)
Operation: Returns true if x is odd, else false
Type of x: integer
Type of result: boolean

EOLN(x)
Operation: Returns true if the end of a line in the file has been reached
Type of x: text
Type of result: boolean

EOF(x)
Operation: Returns true if the end of the file has been reached.
Type of x: file
Type of result: boolean

Transfer functions

TRUNC(x) Operation: Truncates a real value to its integer part
Type of x: real
Type of result: integer

ROUND(x) Operation: Rounds a real value to the nearest integer
Type of x: real
Type of result: integer

ORD(x) Operation: Returns the ordinal number of x.
Type of x: any ordinal type
Type of result: integer

CHR(x) Operation: Returns the character whose ordinal number is x
Type of x: integer
Type of result: char

LOCATION(x) Operation: Returns the address of variable x
Type of x: any type (may also be a procedure name)
Type of result: integer

SIZE(x) Operation: Returns the size of type x in bytes
Type of x: any type identifier
Type of result: integer

HB(x) Operation: Returns the high byte of x
Type of x: integer
Type of result: integer

LB(x) Operation: Returns the low byte of x
Type of x: integer
Type of result: integer

Data transfer procedures

PACK(a,i,z) Operation: Copy the unpacked array a into the packed array z. If the dimension of a is m..n and the dimension of z is u..v and n-m > v-u then the operation is equivalent to:
for j:= u to v do z[j] := a[j-u+i]

UNPACK(z,a,i) Unpacks the above array.

Dynamic allocation procedures

NEW(p) Allocates a new variable v and assigns the pointer reference of v to the pointer variable p. Tag field values may appear as parameters to NEW but are non-functional.

DISPOSE(p) Releases the storage occupied by the variable pointed to by p.

Other functions

SUCC(x) Operation: Returns the successor of x which is next higher value in the enumeration of which x is a member
Type of x: any ordinal type
Type of result: same type as x

PRED(x) Operation: Returns the predecessor of x which is the next lower value in the enumeration of which x is a member
Type of x: any ordinal type
Type of result: same type as x

Other procedures

ESCAPE Causes termination of a block just as if the block end had been reached. If the block is a procedure or function, then control returns to the calling block. If the block is the program block, then program execution is terminated.

note : IF files are declared locally within a procedure, then the files must be closed using the procedure CLOSE before calling ESCAPE. Normal termination of a block results in files automatically being closed.

INPUT AND OUTPUT

Input and output is the communication of a program to the external environment. A program communicates to the external environment through the use of logical files. Logical files are the variables in a program which are declared as type FILE or TEXT. The logical files are then associated with physical files. Physical files are the actual devices of the computer system. A physical file could be a disk file, a terminal , a printer, or some other device. The method of associating logical files to physical files is discussed in the System Implementation Manual.

Predeclared procedures and functions are provided for handling input and output. These procedures and functions have a characteristic unlike other procedures and functions. The number of parameters passed to them can vary. They may be called with no parameters or with several parameters. Since each input and output routine performs an operation on a file, it must know which file to operate on. If a routine is passed the logical file name, then it operates on the specified file, otherwise it operates on a default logical file. The two predeclared variables INPUT and OUTPUT are the default logical files. They are both declared as type TEXT. The one used as the default depends on the routine called. The input routines default to INPUT and the output routines default to OUTPUT.

I/O Routines

Procedures			Functions
input	output	general	EOF
			EOLN
RESET	REWRITE	CLOSE	
READ	WRITE		
READLN	WRITELN		
	PAGE		
	MESSAGE		

A file has associated with it a file pointer. The file pointer is used to point to an individual component of a file. There are two predeclared boolean functions which may be used to check the status of a files pointer. Both functions may or may not take a logical file name as a parameter. If no file parameter is passed, the default is INPUT. The function EOF(file) returns the value TRUE if the pointer is at the end of the file. Otherwise, the value returned is FALSE. The function EOLN(file) can only be used with files of type TEXT. It returns the value TRUE when the files pointer is at the end of a line. Otherwise, the value returned is FALSE.

Syntax of function EOF or EOLN: (default: file = INPUT)

Examples of using EOF and EOLN:

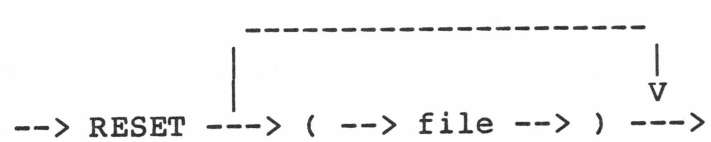
```
WHILE NOT EOF(datain) DO
  BEGIN
  WHILE NOT EOLN(datain) DO
    BEGIN
    READ(ch);
    .....
    END;
  .....
  END;
```

```
IF EOF THEN quit
ELSE
  READ(number);
```

A. RESET

The RESET procedure opens a file so that it can be read. No input can be received from a file without this operation first being performed.

Syntax of RESET: (default: file = INPUT)



The procedure positions the file pointer to the beginning of the file. If the file is empty, then the function EOF(file) becomes TRUE. If the file is not empty, then the function EOF(file) becomes FALSE.

The statement RESET(INPUT) is implicitly executed at the beginning of a program unless the NO INOUT compiler option is used. Therefore, it is not necessary for a program to explicitly open the default logical file INPUT.

Example use of RESET:

```
PROGRAM readdata;
VAR    datain : TEXT;
BEGIN
  RESET(datain);    (*open file datain for reading*)
  .....
END.
```

Input and output to files is buffered. This is to prevent having to access a physical device every time an operation is performed. Each file used by a program has an associated buffer. Unlike standard Pascal, the input buffer of a file is not filled when a reset is performed. The input buffer becomes filled the first time a READ, READLN, EOLN, or EOF is performed on the file. This prevents the normal problems associated with reading from a terminal. Programs can have their logical files remapped from a disk file to a terminal without modification to the program itself.

(See the System Implementation Manual for a description of how to associate logical files to physical files)

B. REWRITE

The REWRITE procedure opens a file so that it can be written. No output can be sent to a file without this operation first being performed.

Syntax of REWRITE: (default: file = OUTPUT)

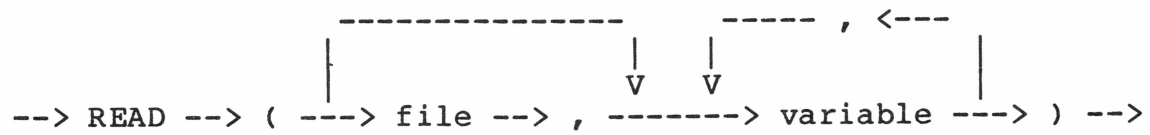
The procedure positions the file pointer to the beginning of the file. The file becomes empty when this happens. This means that any data in the file is lost.

The statement REWRITE(OUTPUT) is implicitily executed at the beginning of a program unless the NO INOUT compiler option is used. Therefore, it is not necessary for a program to explicitly open the default logical file OUTPUT.

C. READ

The READ procedure assigns the value of components of a file to variables.

Syntax of READ: (default: file = INPUT)



The number of variables passed to the procedure determines the number of components read from the file. The components refer to the way the file is logically separated into individual data elements. Each component is of some data type which defines its size. Reading begins with the component pointed to by the file pointer. The first variable specified is assigned the value of this component and then the file pointer is advanced to the next component. This process is continued until all the variables specified are assigned values. The type of each variable must match the type of the file component being assigned to it.

Text files

If the file is of type TEXT, the variables can be type REAL, INTEGER, subrange of integer, CHAR, or strings. Strings are declared as single dimensioned packed arrays of the type CHAR. These types can be intermixed as components of text files. Then they may be read by specifiying variables which match in type and order, the components of the file.

Note: The following characters have special meaning in a text file and may not be read as single characters. Use FILE OF CHAR to avoid this special processing.

HT	-->	#09
LF	-->	#0A
CR	-->	#0D
SUB	-->	#1A

If the variable is of type CHAR, then a single character is read from the file. If the variable is an array of CHAR, then the dimension of the array determines the number of characters read from the file. If an end of line or file mark is encountered before the array is full, then the characters read up to that point are left justified in the array and the remaining elements are filled with blanks. Integer and real numbers are represented in files as strings of characters. Individual numbers in a file are separated by blanks or by an end of line mark. When a number is read, the character string representing the number is automatically converted to its real or integer value before being assigned to the variable. With text files, consecutive read operations automatically skip end of line marks when reading integer, real, or boolean variables. When reading character or string variables, the end of line mark is not skipped. In this case, the procedure READLN must be executed to cause the file pointer to advance to the next line.

Example use with text files:

Consider the following file of data:

```
SAM JONES     25     183.5     369
MARY SMITH    23     105.4     356
      ......
      ......
```

and the declarations:

```
VAR       name             : PACKED ARRAY[1..10] OF CHAR;
          number, total    : INTEGER;
          score            : REAL;
          students         : TEXT;
```

If the file pointer of "students" points to the beginning of a line (it does immediately after a RESET) then:

```
READ(students,name,number,score,total)
```

would assign a string, integer, real, and integer value to the 4 specified variables. The file pointer would then point to the character immediately following the last value read.

Non-text files

If the file is not of type TEXT, then all components of the file are of the same type. The components of a file may be declared to be of any type except the type FILE or structured types containing a component of type FILE. This means for example, that you could declare a file of records. Then an entire record can be read into a variable of the same record type. This however, requires that the file of records has previously been created through the use of the procedure WRITE. The reason for this is that all files which are not of type TEXT are read and written in binary form.

Example use with non-text files:

assume the following declarations:

```
TYPE     food = RECORD
                fruit     : (orange, grape, apple);
                vegetable : (corn, okra, beans);
                cost      : INTEGER;
                END;

VAR      groceries : FILE OF food;
         item      : food;
```

---

then:

```
         READ(groceries, item)
```

would assign one record from the file to the variable "item".

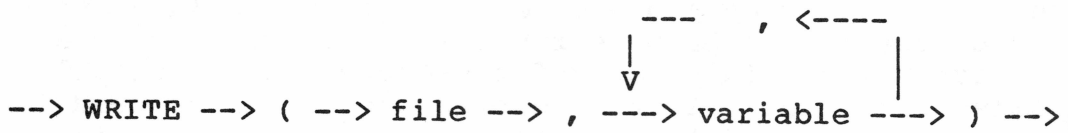
Care should be taken not to read past the end of a file. The function EOF is provided for preventing this from occuring. The program will not abort if you try to read past the end of file, but the value assigned to the variable will be some unknown value.

D. WRITE

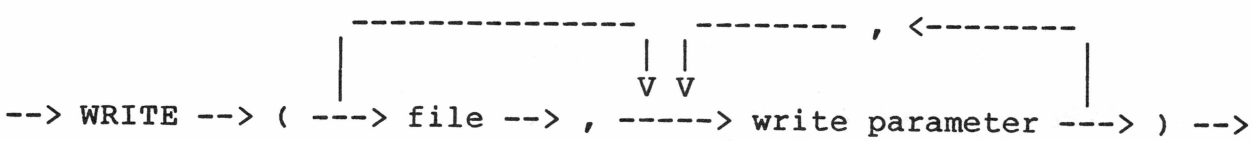
The procedure WRITE appends values to a file. The number of values passed to the procedure determines the number of values output to the file. If a file is declared as type TEXT, then output values can be specified as strings or expressions. If a file is declared as a type other than type TEXT, then the output values are restricted to variables of the same type only.

Syntax of WRITE:

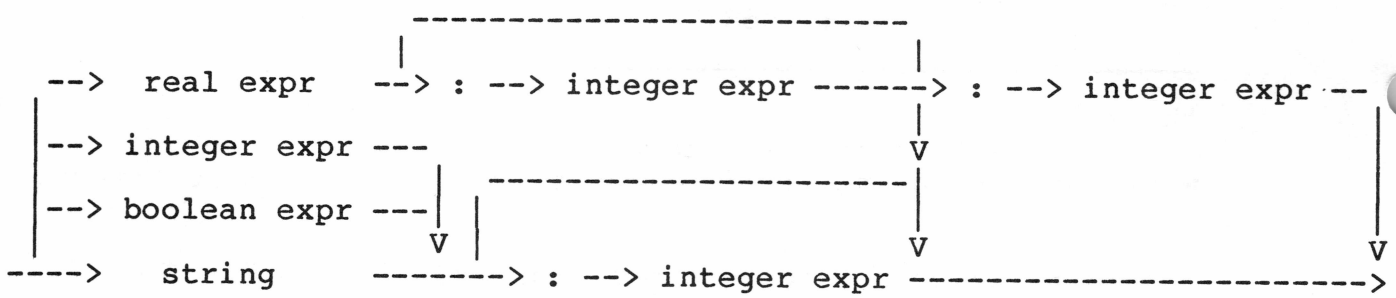
For non-text files:



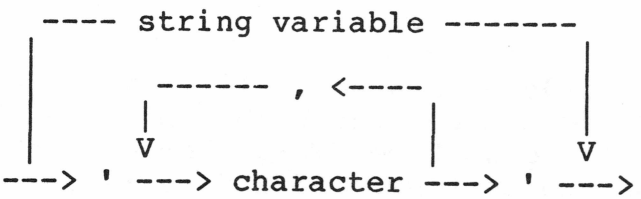
For text files: (default: file = OUTPUT)



Syntax of write parameter:



Syntax of string: (string variable = packed array of char)



Text files

If the file is of type TEXT, then the values output to the file may be specified as strings or as boolean, integer, or real expressions. If a string is specified, then the characters of the string are output to the file. If a boolean expression is specified, then either the characters 'TRUE ' or 'FALSE' are output to the file depending on the value of the expression. If an integer or real expression is specified, then the value of the expression is converted to a character string before being output to the file. An integer expression may be output in hexadecimal or decimal base representation.

The number of characters to output for a value can be specified by an integer expression which follows the value, separated by ":". If the number of characters is not specified for a particular value, then a default number of characters will be output.

---For a string---

If the number is less than the length of the string, then all the characters of the string are output. If the number is greater than the length of the string, then blanks will be appended to the string. The default number is the length of the string.

```
Example:   WRITE(' literal string' : 20)
```

---For a boolean expression---

The same rule applies for the strings 'FALSE' and 'TRUE'.

```
Example:   WRITE(a AND b : 10)
```

---For an integer expression---

If the number is less than the number of digits in the integer, then all the digits are output. If the number is greater than the number of digits, then the excess characters are output as blanks before the integer is output. The default number of digits for integers is 8. An integer value may be written in hexadecimal base format by specifying : width HEX

```
Example:   WRITE(outfile, n+5 :i, j :4 HEX)
```

---For a real expression---

Two numbers may be specified for real values. The first number specifies the total field width. The second specifies the number of digits after the decimal point. If both are specified, the the number will be written in fixed format. Otherwise, the number will be written in exponential format. The default field width for single precision is 12. The double precision default is 20. The maximum field width is 32.

```
Example:   WRITE(2.5*random :5, random/x:9:6)
```

Non-text files

If the file is not of type TEXT, then output values must be variables. Output directed to non-text files is in binary form. This means that values are output in the same form as they are stored. For example, an integer is not converted to a character string before it is output.

Example use with non-text files:

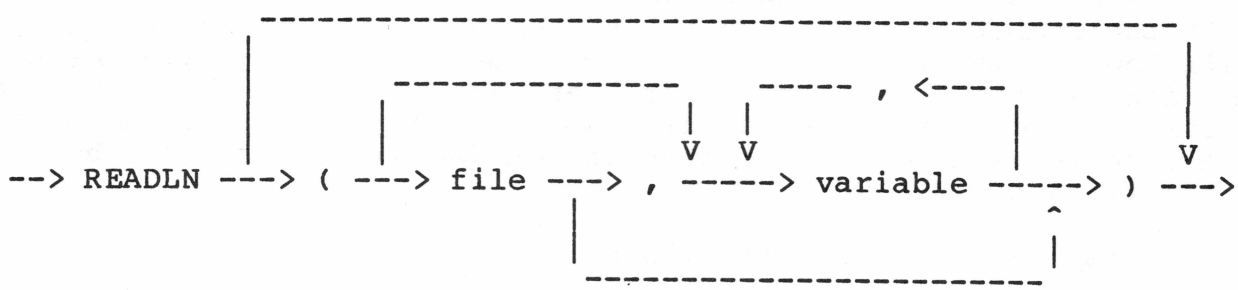
```
WRITE(groceries,item)
```

E. READLN

This procedure can be used only with files of type TEXT. (See section C.1 of chapter 4 for a description of text files.)

The READLN procedure is similar to the READ procedure. The difference is that at the end of the read operation, the file pointer is advanced to the beginning of the next line.

Syntax of READLN: (default: file = INPUT)



The READLN procedure may be called without passing any variables to be read. When no variables are specified, then the procedure just advances the line pointer to the beginning of the next line.

The statement: READLN(var1,var2,var3)
is equivalent to: BEGIN READ(var1,var2,var3); READLN END

The function EOLN can be used to determine whether or not a files pointer is at the end of a line.

Example use of READLN:

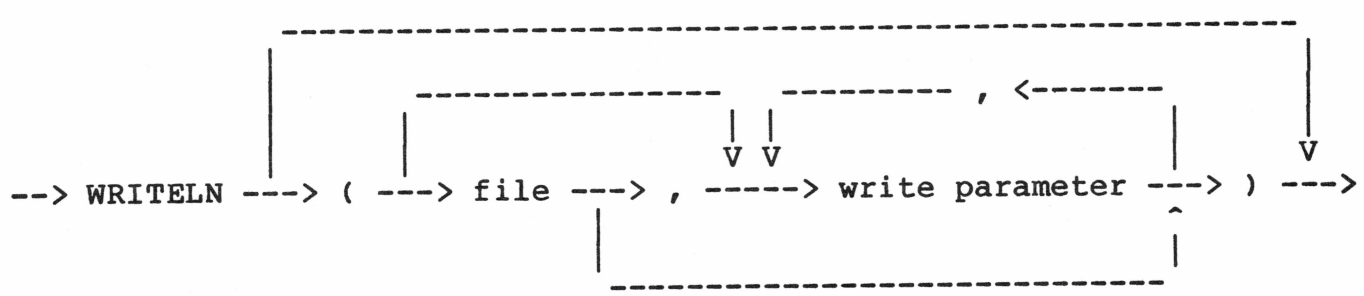
```
i := 0;
WHILE NOT EOF DO
  BEGIN
  i := i+1;
  READLN(a[i]) (*reads one value from each line*)
  ......
  END;
```

```
WHILE NOT EOF (infile) DO
  BEGIN
  WHILE NOT EOLN(infile) DO
    BEGIN
    READ(infile,ch);
    .....
    END;
  READLN(infile);  (*advances file pointer to next line*)
  END;
```

F. WRITELN

This procedure can only be used with files of type TEXT. (See section C.1 of chapter 4 for a description of text files).
The WRITELN procedure is similar to the WRITE procedure. The difference is that at the end of the write operation, an end of line mark is appended to the file.

Syntax of WRITELN: (default: file = OUTPUT)



(See WRITE for syntax of write parameter)

The WRITELN procedure may be called without passing any values to written. When no values are specified, then the procedure just appends an end of line mark to the file.

The statement: WRITELN(var1,var2,var3)
is equivalent to: BEGIN WRITE(var1,var2,var3); WRITELN END

Example use of WRITELN:

```
(*writes 2 values on each line*)
FOR k := 1 TO 100 DO WRITELN(a[k],b[k]);
```

```
FOR j := 1 TO maximum DO
  BEGIN
  i := 0;
  REPEAT
    i := i+1;
    WRITE(number[j]);
  UNTIL (i=5) OR (number[j]>100);
  WRITELN;              (*advance file pointer to next line*)
  END;
```

G. CLOSE

The use of the CLOSE procedure will assure that file data will not be lost if the program abnormally terminates and does not properly close the file. The CLOSE procedure must be used with files which are components of structured variables. (see the appendix)

Syntax of CLOSE:

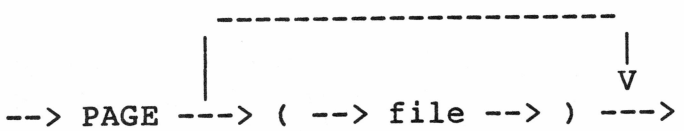
--> CLOSE --> (--> file -->) -->

H. PAGE

The PAGE procedure appends a formfeed to a file. Formfeeds cause printers to skip to the top of the next page. This procedure provides a way of controlling the number of lines printed on a page.

This procedure may only be used with files of type TEXT.

Syntax of PAGE: (default: file = OUTPUT)



I. MESSAGE

The procedure MESSAGE may be used to output strings to the terminal. It takes one parameter which is either a string constant or variable. A string constant is a sequence of characters enclosed in single quotes. A string variable is a variable declared as a packed array of characters.

Syntax of MESSAGE:

Programs which require only string output to the terminal can use this procedure rather than the WRITE procedure.

Example use of MESSAGE:

```
MESSAGE(' time to quit');

MESSAGE(string);
```

APPENDIX

A. COMPILER OPTIONS

Compiler options are provided to change the behavior of the Pascal compiler. These options allow features to be enabled or disabled and can alter the code generated at compile time.

Compiler options are specified in comments. A comment that contains a dollar sign as the first character specifies an option. All compiler options have two states, on and off. An option is turned on by placing its name after the dollar sign. If the option name is preceded by the word "NO", then the option is turned off. Except where noted, the options may appear any place in a program.

DOUBLE

This option specifies that all real variables within the program should be double precision. This option must precede the program statement. If it occurs anywhere else in the program, it will be ignored. If the option is off (the default), then real variables are single precision.

Example:

```
(*$DOUBLE*)
PROGRAM DBL;
VAR
   R : REAL;
BEGIN
END.
```

In this program, the variable "R" will be declared as double precision.

FORDECL

This option is used to change the behavior of loop counters in FOR statements. If the option is turned on(default is off), then all FOR loop counters are treated as temporary variables. They do not need to be declared, and even if a declaration is present, a new variable is used rather than the declared variable. These FOR loop counters are defined only within the loop and disappear when the loop is exited.

Example:

```
PROGRAM FORLOOP;
(*$FORDECL*)
VAR
   A,I : INTEGER;
BEGIN
   A := 0;
   I := 0;
   FOR I := 0 TO 4 DO A := A + I;
   WRITELN(OUTPUT,I,A);
END.
```

In the above program, the I used as a FOR loop counter is a different variable from the I declared in the VAR section. When the write statement is executed, the values 0 and 10 will be printed.

INOUT

This option enables the predeclared files: INPUT and OUTPUT(default is on). If this option is turned off before the PROGRAM statement, then the files input and output will not be declared. This option prevents the reset of INPUT and the rewrite of OUTPUT and can be used to avoid the prompts "INPUT =" and "OUTPUT =" when a program is run.

Example:

```
(*$NO INOUT*)
PROGRAM NOPROMPTS;
BEGIN
  MESSAGE('I WAS NOT PROMPTED FOR INPUT AND OUTPUT')
END.
```

IF

The if option provides conditional compilation. The word IF is followed by the name of a boolean constant. If the constant has the value "TRUE", then compilation continues as if the option had not been present. If the constant has the value "FALSE" then compilation stops at that point, and all text is treated as comments until a (*$NO IF*) is encountered. Note that IF options do not nest. That is, an IF option should not occur within the scope of another if option. The if option can be used to configure a program for different environments with minimum changes to the source. It is also useful for removing debugging statements once the program is working properly.

Example:

```
PROGRAM Test;
CONST
   debug = false;

FUNCTION FACTORIAL(I : INTEGER) : REAL;
BEGIN
   IF I = 0 THEN FACTORIAL := 1
   ELSE BEGIN
      (*$IF DEBUG*)
      WRITELN(OUTPUT,'CALLING FACTORIAL(',I-1,')');
      (*$NO IF*)
      FACTORIAL := I * FACTORIAL(I-1);
      END;
END; (* FACTORIAL *)

BEGIN
   WRITELN(OUTPUT,'FACTORIAL(20)=',FACTORIAL(20));
END.
```

In the above program, the write statement within the recursive function FACTORIAL could be turned on during debugging by setting debug to TRUE. Once the program is running, it can be recompiled with debug set to FALSE. The write statement will be effectively removed. In fact, since no code is generated for it, the resulting object program will be shorter. This has the same effect as removing the statement with an editor or placing open and close comments arround it. The advantage is that many statements can be disabled or enabled with a single change to the source program. Also, it is simple to reenable debugging statements should it become necessary in the future.

NULLBODY

The nullbody option is used to disable code generation for a procedure, function or program. The nullbody option should occur after the BEGIN that starts the block and before any executable statements. Nullbody will prevent code from being generated and can be used when procedures are being compiled separately. Since every program must have a program statement and a main program body, it is necessary to use nullbody to disable code generation for the main program when a subroutine library is being compiled.

For example:

```
PROGRAM SUBLIBRARY;
TYPE
   STRING = PACKED ARRAY[1..80] OF CHAR;

PROCEDURE CONCATENATE(VAR S1, S2, RESULT: STRING);
BEGIN
 (* BODY OF CONCATENATE *)
END;

PROCEDURE MID$(VAR S : STRING; FIRST, LAST : INTEGER;
          VAR RESULT : STRING);
BEGIN
 (* BODY OF MID$ *)
END;

BEGIN
  (*$NULLBODY*)
END.
```

If the above program is compiled, the object file will contain code only for the two procedures: CONCATENATE and MID$. There will be no main program. This allows these procedures to be linked to another program.

INCLUDE

The include option is used to specify within a program, the name of a file which contains Pascal statements which you want included in the compilation process. When the compiler encounters an include option, it opens the specified file and compiles all the Pascal source in the file before continuing compilation of the current file. The include option allows you to include commonly used routines or declarations in a program without actually having the code present. You simply tell the compiler the name of the file containing the Pascal statements and it will include those statements as it compiles.

The include options may be nested. That is, you may include a file which also contains an include compiler option. There is no limit to the number of nested includes. However, the the compiler must maintain a file descriptor for each file that is open at any given time. The file descriptors are allocated memory from the heap. If too many files are open at a time, the compiler may run out of heap during the compile process.

Example use of the INCLUDE option:

```
PROGRAM sample;
{DECLARE contains the declarations for this program}
(*$INCLUDE 'DECLARE'*) {note the quotes}
BEGIN
  {BODY contains the statement body for this program}
  (*$INCLUDE 'BODY'*)
END.
```

LIST

The list option allows you to turn the compiler listing on and off within a program. The default is on. Therefore, the compiler will by default generate a listing which contains all the lines of a program. If it is desired to discard some of the lines of a program from the compiler generated listing, (*$NO LIST*) may be used to tell the compiler to discard all subsequent lines of the program from the listing. The compiler does not stop compiling subsequent lines, it just does not output them to the listing. Object code is still generated. If you wish to turn the compiler listing back on, then (*$LIST*) tells the compiler to start outputting all subsequent lines to the listing again.

The LIST option may be useful when compiling frequently used routines which you know will compile correctly. It provides a method to shorten compiler listings, saving paper when printing, and making it easier to locate other procedures or functions by uncluttering lengthy program listings.

Example using the compiler LIST option:

```
PROGRAM sample;
VAR ...
  PROCEDURE useoften;
  (*$NO LIST*)     {turn off listing for useoften}
  VAR ...
  BEGIN
   ...
  END;             {end of procedure useoften}
(*$LIST*)          {turn listing back on for program}
BEGIN              {beginning of main program}
  ...
END.
```

PAGESIZE

The listing generated by the compiler has printer control information (formfeed) between each page. The compiler outputs a formfeed (hex 0C) to the listing every 62 lines. The formfeed causes most printers to advance the paper to the top of the next page. The PAGESIZE option allows you to change the number of lines that the compiler will output to the listing between formfeeds. The actual number of lines output between formfeeds is 2 more than the number specified by the PAGESIZE option. This is to allow for the heading.

Most operating systems control paging when outputting data to a line printer. The operating system itself maintains a line counter and outputs a formfeed to the line printer after so many lines have been sent to the printer. A command is typically provided to set the number of lines per page or to turn paging control off entirely. If the operating system is controlling paging, the listing generated by the compiler may not be paged properly (ie. the compiler heading may not appear at the top of each page). The number of lines per page used by the operating system should be equal to the number of lines per page used by the compiler, or the operating system paging must be turned off, if compiler generated listings are to be printed properly.

Example use of the compiler PAGE option:

```
(*$PAGESIZE 50*)      {set the number of lines/page to 50}
PROGRAM sample;
{the operating system paging should be set to 52
 or be turned off entirely}
...
BEGIN
  ...
END.
```

WIDELIST

The compiler now generates line numbers for each line of a listing. The WIDELIST option is used to specify that you want the compiler to additionally generate hexadecimal addresses which show the location of the object code for a particular line relative to the start of the procedure, function, or program in which the line appears. This information is useful when used in conjunction with the linking loader to determine the location within a program of a fatal error. You may use the S command of the linking loader to display the starting address of each routine loaded. Then use the R command to run the program. When the program terminates with a fatal error, the absolute hex address of the error is displayed. You may use this address along with the addresses displayed by the S command to determine in which routine the error occurred. By subtracting the address of the error from the starting address of the routine in which the error occurred, you obtain the relative address of the error within that routine. This address corresponds to the address printed on the listing.

Example use of the compiler WIDELIST option:

```
(*$WIDELIST*)  {tell the compiler to print hex addresses}
PROGRAM sample;
...
BEGIN
  ...
END.
```

RANGECHK

A common error which occurs in programs which utilize arrays is to index the array with a value which is outside the array bounds (eg. an array with bounds 1..10 is indexed with the value 11). A common error in programs which utilize subranges is to assign a value which is outside the subrange (eg. a variable is declared as type 0..255 and is assigned the value 275). A common error in programs which utilize enumerations is to increment or decrement past the first or last value of the enumeration (eg. SUCC(color) is executed when color is equal to blue and color is of type (red, green, blue)).

All of these errors may be trapped, causing an appropriate runtime error message to be displayed when such an error occurs during program execution. The RANGECHK option tells the compiler to generate extra code to detect and report errors of the above type when the compiled program is executed.

Since the RANGECHK option does cause additional object code to be generated, you should generally use it only during the debugging stage of program development. The RANGECHK option may be turned on and off throughout a program. The default is off. The IF compiler option may be used to conditionally turn the RANGECHK option on and off as needed for debugging purposes. Example use of the compiler RANGECHK option:

```
PROGRAM sample;
VAR     A,B : ARRAY[1..200] OF CHAR;
        J,K : INTEGER;
        ...
BEGIN
  WRITE(OUTPUT,'Enter size of array: ');
  (*$RANGECHK*)          {turn range checking on}
  ...
  FOR K := 1 TO J DO A[K] := B[K+1];
  (*$NO RANGECHK*)       {turn range checking off}
  ...
END.
```

Note: The RANGECHK option will not detect an error on subrange variables which are assigned invalid values via a read statement. To trap these errors, you must assign the read in value to a subrange variable.

```
READ(VALUE);
SUBRANGE_VARIABLE := VALUE;
```

A common error which occurs in programs which utilize dynamic pointer variables is the inadvertent assignment of the value NIL to a pointer and then the subsequent attempt to use the value pointed to in an expression or in an assignment to a static variable. Another common error is the attempt to utilize an uninitialized pointer. An uninitialized pointer may not point to a location within the allocated heap of the program. It may point into the executing code of the program, making it possible to write data over the instructions, causing very unpredictable results.

The PTRCHECK option is used to tell the compiler to generate extra code in the compiled program to detect and report either of the above types of errors when the program executes. This extra code causes the program to terminate and display an appropriate error message when an invalid use of a pointer variable is detected. The PTRCHECK option may be turned on and off throughout a program. The default is off.

Example use of the compiler PTRCHECK option:

```
PROGRAM sample;
TYPE  customer = RECORD name,add : ARRAY[1..9] OF CHAR END;
VAR   cust     : ^customer;
BEGIN
  (*$PTRCHECK*)
  ...
  WHILE cust<>NIL DO
    ...
END.
```

B.1 Compiler Error Codes

```
  2 IDENTIFIER EXPECTED
  3 'PROGRAM' EXPECTED
  4 ')' EXPECTED
  5 ':' EXPECTED
  6 ILLEGAL SYMBOL
  8 'OF' EXPECTED
  9 '(' EXPECTED
 10 ERROR IN TYPE
 11 LEFT BRACKET '[' OR '(.' EXPECTED
 12 RIGHT BRACKET ']' OR '.)' EXPECTED
 13 'END' EXPECTED
 14 ';' EXPECTED
 15 INTEGER EXPECTED
 16 '=' EXPECTED
 17 'BEGIN' EXPECTED
 20 ',' EXPECTED
 22 '..' EXPECTED
 23 '.' EXPECTED
 49 'ARRAY' EXPECTED
 50 CONSTANT EXPECTED
 51 ':=' EXPECTED
 52 'THEN' EXPECTED
 53 'UNTIL' EXPECTED
 54 'DO' EXPECTED
 55 'TO'/'DOWNTO' EXPECTED
 57 'FILE' EXPECTED
 58 INVALID OR MISSING OPERAND IN AN EXPRESSION
 62 DECIMAL PLACE ALLOWED ONLY FOR REAL
 66 TYPE IDENTIFIER EXPECTED
 80 OPEN COMMENT WITHIN A COMMENT
 81 UNKNOWN OPTION
 82 # REQUIRES A 2 CHARACTER HEX VALUE OR ##
101 IDENTIFIER DECLARED TWICE
102 LOWER BOUND EXCEEDS UPPER BOUND
103 IDENTIFIER IS NOT OF APPROPRIATE CLASS
104 UNDECLARED IDENTIFIER
105 CLASS OF IDENTIFER IS NOT VARIABLE
107 INCOMPATIBLE SUBRANGE TYPES
113 ARRAY BOUNDS MUST BE SCALAR
117 UNSATISFIED FORWARD REFERENCE TO A TYPE IDENTIFER OF A POINTER
119 ';' EXPECTED (PARAMETER LIST NOT ALLOWED)
120 FUNCTION RESULT MUST BE SCALAR, SUBRANGE, OR POINTER
123 FUNCTION RESULT EXPECTED
126 IMPROPER NUMBER OF PARAMETERS
127 TYPE OF ACTUAL PARAMETER DOES NOT MATCH FORMAL PARAMETER
129 TYPE CONFLICT OF OPERANDS IN AN EXPRESSION
132 COMPARISON WITH '>' OR '<' NOT ALLOWED ON SETS
134 ILLEGAL TYPE OF OPERANDS
135 TYPE OF EXPRESSION MUST BE BOOLEAN
```

136 SET ELEMENT TYPE MUST BE SOME ENUMERATION TYPE
138 TYPE OF VARIABLE IS NOT ARRAY
140 TYPE OF VARIABLE IS NOT RECORD
141 TYPE OF VARIABLE IS NOT POINTER
148 SET BOUNDS OUT OF RANGE
152 NO SUCH FIELD IN THIS RECORD
154 ACTUAL PARAMETER MUST BE A VARIABLE
156 MULTIDEFINED CASE LABEL
161 PROCEDURE OR FUNCTION ALREADY DECLARED AT A PREVIOUS LEVEL
165 LABEL ALREADY DEFINED
167 UNDECLARED LABEL
168 LABEL NOT DEFINED
182 "FOR" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
183 "CASE" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
184 "FOR" VARIABLE MUST BE LOCAL
185 OPERATION DEFINED FOR TEXT ONLY
186 OPERATION NOT DEFINED FOR TEXT FILES
193 ACCESS STATEMENT MISSING FOR COMMON
199 FEATURE NOT IMPLEMENTED
202 STRING CONSTANT CANNOT SPAN LINES
203 INTEGER CONSTANT TOO LARGE
210 FIELD WIDTH MUST BE INTEGER
211 FRACTION LENGTH MUST BE OF TYPE INTEGER
212 HEX FORMAT ALLOWED ONLY FOR TYPE INTEGER
219 PARAMETER MUST BE OF TYPE FILE
220 PARAMETER MUST BE OF TYPE INTEGER
223 PARAMETER MUST BE OF TYPE POINTER
230 ILLEGAL TYPE OF PARAMETER IN STANDARD PROCEDURE CALL
250 TOO MANY NESTED SCOPES - LIMIT IS 15
401 OPEN COMMENT ENCOUNTERED IN A COMMENT
403 TO MANY PROCEDURE NESTING LEVELS
404 ARRAY BOUNDS MUST BE SCALAR

B.2 Runtime Error Codes

```
01) OUT OF STACK
    cause: insufficient amount of stack available
    cure : If compiling
           with PASCAL  : switch to PASCALB
           with PASCALB : specify more stack space
                          PASCALB <stack> file
           If executing
           with RUN     : specify more stack space
                          RUN file stack
           with /CMD    : specify more stack space when using B
                          command of LINKLOAD

02) OUT OF HEAP
    cause: insufficient amount of heap available
    cure : If compiling
           with PASCAL  : switch to PASCALB
           with PASCALB : specify less stack space
           If executing
           with RUN     : specify less stack space
           with /CMD    : specify less stack space when using B
                          command of LINKLOAD

03) BAD POINTER
    cause: damaged object file or error in program which causes
           executing code to be overwritten with data
    cure:  If executing one of the system /CMD files:
                restore defective /CMD file from the original master
                disk.
           If executing a user written program:
                recompile the program using the RANGECHK and
                PTRCHECK options and execute once again.  Invalid
                array indexing and most invalid pointer referencing
                will be trapped.  If a range or pointer error
                message is displayed, locate and fix the programming
                error.

04) BAD LEVEL
    see error 03

05) DIVIDE BY 0
    cause: an integer or real divide operation with a divisor of 0
    cure : prevent divisor from becoming 0

06) UNDEFINED PCODE
    see error 03

07) INVALID SET
    cause: set operation results in set with more than 256 members
    cure : restrict set operations to 256 member sets

08) BAD RUNTIME CALL
    see error 03
```

09) IO ERROR
    cause: 1 - file does not exist
           2 - disk is full
           3 - bad disk or hardware
    cure: 1 - specify correct file name
           2 - clear some space on the disk
           3 - run diagnostics

0A) SET ELEMENT TOO LARGE
    cause: attempt to assign an ordinal value > 256 to a set
    cure: limit sets to 256 members

10) RANGE CHECK
    cause: invalid array index, subrange value, or enumeration value
    cure: correct invalid array indexing and/or invalid values

11) BAD DIGIT IN NUMBER
    cause: attempt to read or DECODE an invalid number
    cure: make sure all numbers read or decoded are legal numbers

12) PUT ERROR
    cause: attempt to output an undefined file buffer variable
    cure: assign a proper value to the file buffer variable

13) OVERFLOW
    cause: a real arithmetic calculation overflows
    cure: limit real numbers to the maximum size

15) UNDERFLOW
    cause: a real divide operation causes underflow
    cure: limit real numbers to the minimum non-zero size

16) LOG NEGATIVE
    cause: attempt to take the natural log of a number <= 0
    cure: log is valid positive numbers only

17) SQRT,X^Y NEGATIVE
    cause: attempt to take the square root of a negative number or
           attempt to raise a negative number to a real power
    cure: square root is valid only for number >=0
          only positive numbers may be raised to a real power

EB) ATTEMPT TO WRITE TO INPUT FILE
    cause: opening an output file using RESET
    cure: open the output file using REWRITE

EC) FILE NOT OPEN
    cause: attempt to read or write an unopened file
    cure: open the file using RESET or REWRITE

ED) ATTEMPT TO READ OUTPUT FILE
    cause: opening an input file using REWRITE
    cure: open the input file using RESET

EE) NO MEMORY FOR FILE BUFFER
    cause: not enough space for file buffer in heap
    cure: execute program using less stack

C. Standard 7-bit USASCII Character Set

Decimal	Octal	Hex	Graphic	Name
0.	000	00	^@	NUL (used for padding) <null>
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm)
8.	010	08	^H	BS (backspace) <bs>
9.	011	09	^I	HT (horizontal tab) <tab>
10.	012	0A	^J	LF (line feed) <lf>
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page) <ff>
13.	015	0D	^M	CR (carriage return) <cr>
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1 (device control 1, XON)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3 (device control 3, XOFF)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)
25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[	ESCAPE (alter mode, SEL) <esc>
28.	034	1C	^\	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^_	US (unit separator)
32.	040	20	"_"	space or blank <sp>
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)
40.	050	28	(	left parenthesis
41.	051	29	)	right parenthesis

42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point)
47.	057	2F	/	(right) slash
48.	060	30	0	numeral zero
49.	061	31	1	numeral one
50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark
64.	100	40	@	atsign
65.	101	41	A	upper-case letter ABLE
66.	102	42	B	upper-case letter BAKER
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICHO
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA

91.	133	5B	[	left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D	]	right square bracket
94.	136	5E	^	uparrow (carat)
95.	137	5F	_	underscore
96.	140	60	`	(single) back quote
97.	141	61	a	lower-case letter able
98.	142	62	b	lower-case letter baker
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo
102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel
105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar
112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whisky
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C	|	vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	<rubout>	DEL <del>

D. Differences from Standard

The standard used is defined by "User Manual and Report", second edition, Jensen and Wirth, Springer-Verlag. The following sections pertain to the differences in Alcor Systems implementation of Pascal as compared to the standard. The extensions are added to provide extra power to the language. All implementations of Pascal by Alcor Systems contain these added features. If a program is to be transported to a computer system using some other implementation of Pascal, these features should not be used in the program.

D.1 Omissions

1) Procedures or functions may not be passed as parameters to other procedures or functions.

D.2 Extensions

1) Common variables which provide a mechanism for statically allocating local variables are implemented through the use two new declaration parts: COMMON and ACCESS.

2) The declaration sections LABEL, CONST, TYPE, VAR, COMMON, and ACCESS may appear any number of times and in any order within a block.

3) The Type Transfer Operator allows variables to be referenced through the use of a type template.

4) Single elements of packed structures may be passed as parameters.

5) The OTHERWISE clause is implemented in the CASE statement. If omitted, and there is no match, execution transfers to the next statement.

6) Identifiers can include the characters '_' and '$'. Also, no distinction is made between upper and lower case letters.

7) Integer constants or characters may be represented in hex.

8) Mixed mode arithmetic is implemented.

9) The procedures READ or READLN will accept string and boolean variables.

10) External procedures or functions may be declared. This feature provides a way of accessing external routines.

11) Input files are not opened until necessary. This eliminates the synchronization problem when doing interactive input from a terminal.

12) Labels may range from -32768 to 32767.

13) Alternate symbols are implemented for brackets and the pointer symbol.

14) The LOCATION function allows the determination of the address of a variable or a procedure.

15) The SIZE function allows the size of a type to be determined.

16) The HB function returns the high byte of an integer variable.

17) The LB function returns the low byte of an integer variable.

18) The procedure MESSAGE provides an additional method for handling string output to a terminal.

19) The procedure CLOSE allows files to be explicitly closed.

20) The procedure ESCAPE allows exiting a block at any point within the block.

21) The type STRING is a predefined dynamic data type. A string function library is provided for use with this data type.

22) Libraries are provided to access the hardware features of the specific machine.

23) Compiler options are provided to control various functions.

D.3 Other Implementation Characteristics

The following is a list of specific implementation decisions which are not defined by the standard.

1) Only the first 8 characters of an identifier are stored. This means that identifier names should be selected such that the first 8 characters form a unique name.

2) There is a limit of 256 elements for sets, enumerations, CASE statement labels, and parameters to a procedure or function.

3) Pascal source is restricted to 80 columns.

4) The association of logical files to physical devices is made either interactively from the terminal or through a procedure call.

The following is a list of characteristics which are slightly altered from the standard.

1) Operator precedence has been altered to eliminate the need for excessive use of parentheses in expressions. The precedence is the same as that used in BASIC. The difference is the precedence assigned to the Boolean operators. The precedence defined by the standard makes the Boolean operator OR equal in precedence with + and -, the Boolean operator AND equal in precedence with *, /, DIV, and MOD, and NOT has the highest precedence of any operator except the parentheses. Parentheses may be used when transportable programs are being written to maintain compatability with the standard. This alteration of precedence should not cause any problems when transferring programs written in standard Pascal to Alcor Pascal.

2) Although structured variables may contain components of type FILE, the I/O routines will accept only simple variable names. Therefore, use of files within structured variables may be used only in a restricted manner.

3) A GOTO statement may not reference a label outside the block in which the statement appears.

E. THE TYPE STRING

The standard Pascal string is defined to be a PACKED ARRAY OF CHAR. Variables of this type are restricted to a predetermined size. (ie. the size of the array must be specified and cannot be altered during program execution). The predefined type STRING is dynamic. The size of a variable declared as type STRING is determined during program execution. Variables of this type may change in size as the program executes. In addition, variables of type STRING may be used in conjunction with a runtime library of string manipulation functions.

Syntax of type STRING:

Example:

```
VAR  str1, str2, str3 : STRING;
```

Assigning values to dynamic string variables

A dynamic string may be created through the use of the predeclared transfer function BLDSTR. This function has one parameter which may be either a variable of the type PACKED ARRAY OF CHAR or a string constant. The function returns a dynamic string of the same length as the array or string constant passed to it.

Example:

```
str1 := BLDSTR('literal string constant');

str2 := BLDSTR(stringconstant);

str3 := BLDSTR(arrayvariable);
```

The procedures READ and READLN have been extended to accept variables of the type STRING. When a variable of type STRING is specified, all characters from the current file pointer to the end of line mark are read. The size of the string is then equal to the number of characters read. If a read is performed while at the end of line mark, the string variable is assigned an empty string. An empty string is a string of zero length.

Example:

```
READ(str1);

READLN(filename,str2);
```

A string variable may be assigned to another string variable. An assignment between two string variables results in both string variables referencing the same string. (ie. both string variables point to the same location in memory)

Example:

```
str1 := str2;
```

NOTE: For most applications, the preferred method of assignment between two string variables is through the use of the library function CPYSTRING. If two string variables point to the same location and one is disposed (using DISPOSE), then both string variables will become undefined.

A string variable may be assigned a string formed by one of the string manipulation functions in the runtime library. For example, there is a function provided which may be used for assignment between two string variables. The function CPYSTR takes a string variable as a parameter and copys it to another location. The string appearing on the left side of the equal sign then references the new location. In other words, instead of having one copy of the string as in the above example, there are now two copies.

Example:

```
str1 := CPYSTR(str2);
```

Outputing dynamic string variables

The WRITE and WRITELN procedures have been extended to accept variables of the type STRING. When a dynamic string is output, the number of characters written is equal to the length of the string.

Converting a dynamic string into an array

Dynamic strings can only be accessed as a whole. (ie. the individual characters of the string cannot be accessed) The predeclared procedure GETSTR will copy a dynamic string variable into a variable of the type PACKED ARRAY OF CHAR. It accepts two parameters. The first parameter is the dynamic string variable. The second parameter is the array variable. The string is left justified in the array. If the string is longer than the array, then it is truncated. If the string is shorter than the array, then the array is padded with blanks.

Example:

```
GETSTR(strl, arrayvariable);
```

Recovering memory used by a dynamic string

The memory used by a dynamic string may be recovered through the use of the standard procedure DISPOSE. When a string variable is passed to the DISPOSE procedure, the memory used by the string is freed and the string variable becomes undefined. In addition, any other string variable which points to the same string will become undefined. Each time a string variable is assigned a value, it points to a new string and the old string is then lost. The memory it uses cannot be recovered. Therefore, before assigning the string variable a new value, the memory used by the old value should be recovered if space is important.

Example:

```
strl := BLDSTR('this is the first value');
...
DISPOSE(strl);
strl := BLDSTR('this is the second');
```

Using the string library

There is a long list of string manipulation functions available in the runtime library. In order for a program to have access to these functions, it must include an external declaration for each function used. A file of external declarations for all the string functions is supplied on disk. The text editor may be used to insert this file into the programs that use these functions. The declarations for any functions which are not used may be deleted if desired. If only one or two functions are used, you may prefer just to type in the external declaration.
(See the System Implementation Manual for a description of the string manipulation functions)

Example use of dynamic strings:

```
PROGRAM sample;

VAR  firstname, lastname,
     space, fullname          : STRING;

FUNCTION CONC(s1,s2 : STRING) : STRING; EXTERNAL;
(*CONC is a string library function which concatenates 2 strings*)

BEGIN
  space := BLDSTR(' ');
  WRITELN(' enter first name');
  READLN(firstname);
  WRITELN(' enter last name');
  READLN(lastname);
  fullname := CONC(CONC(firstname,space),lastname);
  WRITELN(fullname)
END.
```

F. I/O PROCEDURES GET and PUT

File buffer variables and the procedures GET and PUT are I/O features of Pascal which are not often used. The procedures READ and WRITE are abbreviated forms for accomplishing the same I/O tasks. However, file buffer variables do provide a means of performing lookahead in a file. (ie. you may check the value of the next component in a file before actually reading it) The ability to perform lookahead may offer some advantages in certain applications. (eg. the scanner of a compiler)

File Buffer Variables

There is a file buffer variable associated with each file in a program. The buffer variable is used as temporary storage for file components as they are passed to or from the associated file. The buffer variable is the same size and type as an individual component of the file. The individual components of TEXT files are characters. Therefore, the file buffer variable associated with a file of type TEXT has a size of one byte (8 bits) and is of type CHAR. A file declared as FILE OF INTEGER consists of individual components of type INTEGER. The associated file buffer variable will have a size of two bytes (ie. integers require two bytes of storage) and be of type INTEGER.

The buffer variable associated with a particular file may be referenced in the same manner as pointer variables, using either the ^ or @ symbol. The buffer variable of a particular file is referenced by following the logical file name with either of these two symbols. For example, the buffer variable of the logical file INPUT is referenced by either INPUT^ or INPUT@.

Files are opened for reading or writing by the procedures RESET or REWRITE respectively. When a file is opened by RESET, the buffer variable associated with the file is assigned the value of the first component in the file. If the file is empty at the time it is opened, then the value of the buffer variable is undefined. When a file is opened by REWRITE, its associated buffer variable is undefined.

File buffer variables may be assigned values using the assignment statement. For example, OUTPUT@ := 'A' will assign the character A to the file buffer variable associated with logical file OUTPUT. Additionally, the procedures READ, READLN, and GET will alter values of file buffer variables associated with input files. The buffer variables associated with output files become undefined after performing the operation specified by the WRITE, WRITELN, or PUT procedures.

The GET Procedure

The GET procedure assigns the value of the next component of a file to the buffer variable associated with that file. If there is no next component (ie. end of file), then EOF on that file becomes TRUE and the value of the buffer variable is undefined.

Syntax of GET: (default: file = INPUT)

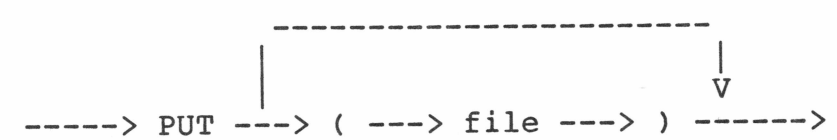
Examples:

```
GET        {assigns the next character of the logical
            file INPUT to the buffer variable INPUT@}

GET(F)     {assigns the next component of the logical
            file F to the buffer variable F@        }

{ READ(f,x)    is equivalent to    x := f@; GET(f) }
```

The PUT Procedure

The PUT procedure appends the value of the buffer variable for a particular file to the end of that file. After the operation, the value of the buffer variable becomes undefined.

Syntax of PUT: (default: file = OUTPUT)



Examples:

```
PUT        {appends the value of the buffer variable
            OUTPUT@ to the end of the logical file OUTPUT}

PUT(F)     {appends the value of the buffer variable F@
            to the end of the logical file F            }

{ WRITE(f,x)   is equivalent to    f@:=x; PUT(f)        }
```

Example use of GET and PUT and file buffer variables:

The following program copies the contents of logical file "infile" to logical file "outfile".

```
PROGRAM filecopy;
VAR     infile, outfile  : TEXT;
BEGIN
  RESET(infile);                {infile@ = first character}
  REWRITE(outfile);             {outfile@ is undefined}
  WHILE NOT EOF(infile) DO
    BEGIN
    WHILE NOT EOLN(infile) DO
      BEGIN
      outfile@ := infile@;      {define outfile@}
      PUT(outfile);             {write outfile@}
      GET(infile)               {get next character}
      END;
    READLN(infile);             {advance to next line}
    WRITELN(outfile)            {advance to next line}
    END
END.
```

G. USING FILES IN STRUCTURED VARIABLES

This implementation of Pascal does not fully support the use of files which are components of structured variables. The following declarations are examples of the use of files in structured variables:

```
VAR files   : ARRAY [1..5] OF TEXT;      {array of files}
    student : RECORD
              name   : ARRAY[1..20] OF CHAR;
              scores : FILE OF INTEGER; {file in record}
              END;
```

The above declarations are legal but the I/O routines (see chapter 10 of reference manual) will not accept file names which are not simple. An example of a simple name is "outfile". With the above declarations, the file names are not simple names. I/O statements like the following would generate compile errors:

```
READLN(files[2],...
WRITELN(files[5],...
READ(student.scores,...
WHILE NOT EOF(student.scores) DO ....
```

For applications that need to use files as components of structures, there is a method of avoiding the simple name restriction to file names. Simply write your own I/O routines which act as interface to the Pascal I/O routines. You may pass non-simple file names to these interface routines which then use simple names in the actual I/O operations. (see the example program on the following page) It is important to note that the file variables should be passed by reference.(preceded by VAR in the parameter list)

When using files which are components of structures, make sure that the following two operations are always performed on the files:

1) Before opening the file, the file must first be initialized. The following operation will initialize a file. The filename may be simple or non-simple when performing this operation.

   ```
   filename::INTEGER := 0   {initializes file "filename"}
   ```

2) Before exiting the program, the file must be explicitly closed by the CLOSE procedure. Failure to do so will probably result in loss of the file.

```
(*$NO INOUT*)
PROGRAM files_in_structures;

(* Sample program which uses array of files *)
(* This program prompts for a file name and
    then sends the file to the line printer *)

VAR     ch     : CHAR;
        files : ARRAY[1..2] OF TEXT;

PROCEDURE openr(VAR filename : TEXT);
BEGIN
  filename::INTEGER:=0;    {initialize file}
  RESET(filename)          {open file for reading}
END;

PROCEDURE openw(VAR filename : TEXT; name : STRING);
  PROCEDURE SETACNM(VAR f : TEXT; name : STRING); EXTERNAL;
BEGIN
  filename::INTEGER:=0;    {initialize file}
  SETACNM(filename,name); {eliminate prompt for filename}
  REWRITE(filename)        {open file for writing}
END;

PROCEDURE closefile(VAR f : TEXT);
BEGIN
  CLOSE(f)         {close file}
END;

PROCEDURE readfile(VAR f : TEXT; VAR data : CHAR);
BEGIN
  READ(f,data)     {read from file}
END;

PROCEDURE writefile(VAR f : TEXT; data : CHAR);
BEGIN
  WRITE(f,data)    {write to file}
END;
```

```
PROCEDURE writeline(VAR f : TEXT);
BEGIN
  WRITELN(f)       {advance to next line of output file}
END;

PROCEDURE readline(VAR f : TEXT);
BEGIN
  READLN(f)        {advance to next line of input file}
END;

FUNCTION endfile(VAR f : TEXT) : BOOLEAN;
BEGIN
  IF EOF(f) THEN endfile := TRUE else endfile := FALSE
END;

FUNCTION endline(VAR f: TEXT) : BOOLEAN;
BEGIN
  IF EOLN(f) THEN endline := TRUE else endline := FALSE
END;

BEGIN
  openr(files[1]);                      {open input file}
  {note to CP/M users --> :L should be changed to LST:}
  openw(files[2],BLDSTR(':L'));         {open output file}
  WHILE NOT endfile(files[1]) DO
    BEGIN
    WHILE NOT endline(files[1]) DO
      BEGIN
      readfile(files[1],ch);
      writefile(files[2],ch);
      END;
    readline(files[1]);
    writeline(files[2])
    END;
  closefile(files[1]);
  closefile(files[2]);
END.
```

H. USING GLOBAL VARIABLES IN EXTERNAL ROUTINES

It is recommended that whenever possible, variables should be passed to routines rather than allowing the routines to access global variables. However, sometimes the use of global variables is necessary. When using global variables in an external routine (ie compiled separately), it is necessary to duplicate the exact global environment when the external routine is compiled. Otherwise, referencing of global variables within the external routine will not be correct.

The use of the compiler INCLUDE option is very helpful to insure that the declarations used in the main program are exactly duplicated in the separately compiled routine. The following example illustrates the use of global variables in a separately compiled procedure.

```
----- file containing the main program -----

PROGRAM main;
{the file GLOBAL contains the declarations for the main program}
{$INCLUDE 'GLOBAL'}
PROCEDURE separate; EXTERNAL;
BEGIN
  letter:='a';
  digit:=10;
  separate;
END.

----- file containing the external procedure -----

PROGRAM compile_separately;
{duplicate the global environment}
{$INCLUDE 'GLOBAL'}
PROCEDURE separate;
BEGIN
  WRITELN('letter = ',letter);
  WRITELN('digit = ',digit:2);
END;
BEGIN
  {$NULLBODY}
END.

----- the file 'GLOBAL' -----

VAR   letter : CHAR;
      digit  : INTEGER;
```

I. USING COMMON VARIABLES

Often when creating libraries, such as a set of graphics routines, it is difficult to avoid the need for using global variables. There is usually a routine which does some initial processing to define variables which are needed by many of the other routines in the library. If these variables are local to the routine, they become undefined when the routine terminates. Of course, these variables could be retained if they were included as parameters to the routine. However, often these variables are not pertinent to the functionality from an end users point of view. Making them a part of the parameter list complicates the use of the library routines. Another alternative is to make these variables global. This is a problem too, because each programmer who uses the library must know of these variables and make appropriate declarations for them. Common variables offer a clean solution to this type of programming problem. They essentially provide the ability to use global variables in libraries without the need for programs which use the library to even be aware of their existence. The following example illustrates the use of common variables.

```
----- file containing library of routines -----

PROGRAM library;
COMMON xscale, yscale : REAL;

PROCEDURE axis(xmin,xmax,ymin,ymax : REAL);
ACCESS xscale, yscale;
BEGIN
   xscale := 512/(xmax-xmin);
   yscale := 256/(ymax-ymin);
END;

PROCEDURE scale(x,y : REAL);
ACCESS xscale, yscale;
VAR  ix,iy : INTEGER;
BEGIN
  ix := ROUND(xscale*x);
  iy := ROUND(yscale*y);
  WRITE('original values: x,y = ',x:6:1,',',y:6:1);
  WRITELN('scaled values: x,y = ',ix:3,',',iy:3);
END;

BEGIN
  {$NULLBODY}
END.

----- user program -----

PROGRAM user;
VAR i : INTEGER;
PROCEDURE axis(xmin,xmax,ymin,ymax : REAL); EXTERNAL;
PROCEDURE scale(x,y : REAL); EXTERNAL;
BEGIN
  axis(0.0,10.0,0.0,5.0);
  FOR i := 0 TO  10 DO scale(i,i/2);
END.
```

ADVANCED DEVELOPMENT PACKAGE

TABLE OF CONTENTS

Introduction..... 1

Chapter 1

Using the Optimizer..... 2
A. When to use it..... 2
B. How to use it..... 2
C. Examples..... 4

Chapter 2

Using the Code Generator..... 5
A. When to use it..... 5
B. How to use it..... 6
C. Examples..... 7

Chapter 3

Mixed Mode Operation..... 9
A. When to use it..... 9
B. How to use it.....10
C. Examples.....10

Chapter 4

System Overview.....16
A. The P-code.....16
B. The Interpreter.....16
C. Runtime Support.....17
D. Memory Map.....17
E. How the Optimizer Works.....18
F. How the Code Generator Works.....18

Chapter 5

System Output.....19
A. Assembly Language 19
B. Assembly Language Structure.....19
C. Assembly Language Format.....20
D. Object Format.....22
E. Splitting Object Files.....23

INTRODUCTION

The advanced development package (ADP) is a software tool which adds a great deal of power and versatility to the TRS-80 Pascal System. The advanced development package consists of two programs. One program is an optimizer which reduces the size of programs. The other is a code generator which increases the speed of programs. The combination gives the programmer the ability to customize each application program, allowing for maximum utilization of the systems capabilities.

The need for the optimizer occurs when writing large programs. All programs require memory to store instructions and memory to store data. Large programs require a lot of memory to store instructions. The memory used for storing instructions subtracts from the memory available for storing data (ie. the more memory used for storing instructions, the less available for storing data). The optimizer's purpose is to reduce the amount of memory used by the instructions in order to make more memory available for storing data.

The need for the code generator occurs when execution speed is important. The compiler translates Pascal source programs to instructions known as p-code. The computer cannot directly execute instructions in p-code form. Instead they are executed by another program known as an interpreter. Maximum execution speed can be achieved by translating programs to machine code (the form which the computer hardware can understand and execute directly without interpretation). The purpose of the code generator is to translate p-code instructions to machine instructions. This provides a method for achieving maximum execution speed.

The addition of the ADP to TRS-80 Pascal provides the programmer with a very flexible language system which offers a unique ability. This is the ability to mix p-code with machine code. P-code has the advantage of compactness while machine code has the advantage of speed. The ability to mix the two makes it possible to customize application programs in order to achieve optimum performance. The bottle neck areas of a program may be translated to machine code for maximum speed while the rest of the program can be left in p-code form. This allows programs to benefit from both compactness and speed.

USING THE P-CODE OPTIMIZER

The optimizer is a program which takes the compiler generated p-code as input and outputs an optimized form of the same p-code. Although optimized p-code will execute faster than non-optimized p-code, the main purpose of the optimization is to make the p-code more compact. The difference in size of the optimized versus non-optimized p-code is dependent on the types of language features utilized by the original source program. Typically, the percent reduction in size due to optimization will fall in the range of 10 to 30 percent. This size reduction is sometimes very important. By making the program smaller, there is more room for data. Often times, it will enable the execution of a program that otherwise would run out of memory.

A. When to Use the Optimizer

The optimizer should be used any time program size is an important factor. A programs memory requirements are determined by the number of executable instructions and by the number and sizes of the variables used. The factor that the optimizer addresses is the number and length of instructions. The greatest benefit will then be realized when optimizing long programs (>1000 lines). However, in many cases optimized code is slightly faster than non-optimized code and even short programs will sometimes benefit enough to make optimization worth while. In addition, if a short program requires lots of data storage, optimization will maximize the amount of memory available for the data.

B. How to Use the Optimizer

Any p-code object file may be used as input to the optimizer program. The compiler uses a /OBJ extension as a default for p-code object files. Whole programs or separately compiled parts of a program may be optimized. In either case, simply compile the Pascal source and then run the compiler generated p-code through the optimizer.

NOTE: Only p-code object files may be optimized. Do not attempt to optimize command files (/CMD) or files generated by the code generator (/COD)

The optimizer program is stored as a command file and therefore may be executed simply by typing OPTIMIZE from the top level of the operating system. Like the compiler, it has two forms for input, a short form and a long form.

The short form:

```
OPTIMIZE filename
```

NOTE: Filename may include a drive specification.
Example: DATABASE:1
When a drive is specified the /OPT file is placed on the same drive as the /OBJ file, otherwise the operating system decides which drive to use.

The filename should not include an extension. The optimizer appends the default extension /OBJ to the file name. The output of the optimizer (the optimized p-code) is placed in a file of the same name but with the extension /OPT. The /OPT file may then be used just as any /OBJ file is used in conjunction with the RUNP and LINKLOAD commands.

The long form:

```
OPTIMIZE
LISTING = listingfile/ext or device (:C,:L,or :D)
INP_OBJ = inputfile/ext
OUT_OPT = outputfile/ext
```

NOTE: File names may also include drive specifiers.
Example: DATABASE/OPT:1

The long form requires that you enter the full file name, including extension, for both the input file (non-optimized) and output file (optimized). The LISTING will show the name of each separate module in the input p-code file as it is processed. After each name will appear its original size in bytes followed by its optimized size in bytes. The LISTING may be directed to a file or device. Typing a carriage return will direct the listing to the CRT.

At completion, the optimizer program will display on the listing the size of the non-optimized p-code used as input and the optimized p-code generated as output.

```
ORIGINAL LENGTH = size in bytes
OPTIMIZED LENGTH= size in bytes
```

C. Example Use of the Optimizer

The following is an example of optimizing the DATABASE/PCL program which was supplied with the TRS-80 Pascal System. The example demonstrates use of the optimizer in both the short and long forms.

step 1 ---> compile the database program

```
PASCAL DATABASE
```

step 2 ---> optimize the p-code in file DATABASE/OBJ

short form example:

```
OPTIMIZE DATABASE
```

long form example:

```
OPTIMIZE
LISTING = :L
INP_OBJ = DATABASE/OBJ
OUT_OPT = DATABASE/OPT
```

Both the short form and long form above would produce the same result. The p-code in file DATABASE/OBJ would be optimized and output to the file DATABASE/OPT. The short form would direct the listing to the CRT while the long form would direct the listing to the line printer. The listing output to the line printer would appear as below.

```
NOCUSTMR  15    13
PRESS     56    49
NEWSPACE  43    29
READDBAS  226   162
WRITEDBA  389   312
CUSTMROU  525   423
READTRAN  269   208
WRITETRA  415   325
DISPLAYD  114   84
LISTTRAN  130   91
LISTCUST   64   50
HEADING    70   64
MAINMENU  743   667
QUERYMEN  462   416
ADDCUSTM  193   153
QUERYTRA  213   161
ADDTRANS  349   266
SEARCHCU  284   218
QUERY     170   132
DATABASE  302   264

     ORIGINAL LENGTH = 5437
     OPTIMIZED LENGTH= 4418
```

USING THE CODE GENERATOR

The code generator is a program which translates p-code instructions to native machine instructions. Any compiler generated p-code object file or optimized p-code file may be used as input to the code generator program. Whole programs or separately compiled parts of programs may be translated (codegened) to machine code to increase execution speed. The speed increase realized from code generation is dependent on the nature of the program. Typically, codegened programs will gain a factor of 3 to 5 times increase in speed over that of pure p-code programs.

A. When to Use the Code Generator

The code generator increases execution speed by translating p-code instructions to machine instructions. Since each p-code is equivalent to several machine instructions, code generation also causes an increase in size. Therefore, the decision of whether or not to perform code generation on a program must not only be based on speed requirements, but also on program size. Typically, code generation will cause the size of the object to increase by a factor of 2 to 3 over that of pure p-code.

The execution speed of most programs will be adequate even when left in p-code form. However, programs which do lots of calculations within loops may benefit significantly through code generation. Also, when a program contains one or more procedures which are frequently called, code generation on these sections of the program can provide quite an improvement in execution speed. For example, the scanner of the compiler is a procedure which reads the text of a pascal program and distributes it to other parts of the compiler. Since it is called frequently, much the of time spent during a compile is inside this one procedure. Code generation of the scanner can increase compile speed significantly. By selecting the parts of a program which most effect speed and performing code generation only on those parts, speed can be increased without significant increase in size.

The determination of whether or not to codegen a program can be made by observation. First run the program in p-code form. If execution speed is observed to be slow, the next step is to determine whether or not to codegen the whole program or selected parts of the program. As a general rule, small programs should be totally codegened. The size increase for small programs will probably be insignificant. However, for large programs, the size increase may be very significant.

For large programs (>1000 lines), a factor of 2 or 3 increase in object size will significantly reduce the amount of memory left for the program data area (stack and heap). In cases where the size increase would not allow enough room for data area, selected procedures should be declared as externals and compiled separately. The procedures selected should be the ones which most effect execution speed. These procedures may then be codegened and linked to the main program. This process will allow for an increase in speed without causing the size to increase to a level that prevents the program from being executed.

The code generator performs most of the optimizations performed by the optimizer. Therefore, it is not necessary to optimize a program before performing code generation.

B. How to Use the Code Generator

Any compiler generated or optimized p-code file may be used as input to the code generator. The compiler generates files with the default extension of /OBJ. The optimizer generates files with a default extension of /OPT. Whole programs or separately compiled programs may be codegened. In either case, simply compile the Pascal source and run the code generator program, using the compiler generated object file as input. Of course, optimized p-code files may also be used as input.

The code generator program is stored as a command file and therefore may be executed simply by typing CODEGEN from the top level of the operating system. Like the compiler and optimizer, it has two forms, a short form and a long form.

The short form:

```
CODEGEN filename
```

NOTE: Filename may also include a drive specifier.
Example: BENCHMK:2
When a drive is specified, the /COD file is placed on the same drive as the /OBJ file, otherwise the operating system decides which drive to use.

The filename should not include an extension. The code generator appends the default extension /OBJ to the file name. The output of the code generator is placed in a file of the same name but with the extension /COD. The /COD file may then be used just as any /OBJ or /OPT object file in conjunction with the RUNP or LINKLOAD commands. However, do not attempt to optimize a /COD file. The /COD files contain machine instructions and the optimizer accepts only p-code instructions.

The long form:

```
CODEGEN
INP_OBJ = inputfile/ext
OUT_COD = outputfile/ext
DO YOU WANT ASSEMBLY LANGUAGE SOURCE? (Y,N): y or n
```

NOTE: File names may also include drive specifiers.
Example : BENCHMK/COD:2

The long form requires that you enter the full file name, including extension, for both input and output files. If assembly language output is desired, answer Y to the last prompt, otherwise answer N. If assembly language output is requested, the following prompt will appear.

```
SOURCE  = file/ext
```

The additional assembly language output will be directed to the file specified. The assembly language output is discussed in chapter 5.

NOTE : The file CODEINIT/DAT must be on line when executing CODEGEN. CODEGEN uses this file for initialization.

C. Example Use of the Code Generator

The following is an example of codegening a program.

step 1 ---> compile BENCHMK/PCL

PASCAL BENCHMK

step 2 ---> codegen the compiled program

short form example:

CODEGEN BENCHMK

The above example uses BENCHMK/OBJ as input and directs the codegened output to file BENCHMK/COD.

long form example 1:

```
CODEGEN
INP_OBJ  = BENCHMK/OBJ
OUT_COD  = BENCHMK/COD
DO YOU WANT ASSEMBLY LANGUAGE SOURCE ? (Y,N): N
```

The above example does exactly the same thing as the short form example.

long form example 2.

```
CODEGEN
INP_OBJ = BENCHMK/OBJ
OUT_COD = BENCHMK/COD
DO YOU WANT ASSEMBLY LANGUAE SOURCE ? (Y,N): Y
SOURCE = BENCHMK/SRC
```

The above example does the same thing as the previous examples except that it also generates an assembly language output which is directed to the file BENCHMK/SRC. The assembly language output is explained in chapter 5.

MIXED MODE OPERATION

Through the use of the linking loader (the LINKLOAD command), pure p-code object (/OBJ) files may be linked with codegened (/COD) files. Executable programs (/CMD files) may then be built which contain mixed instructions, both p-code and machine code. This ability is important when writing large programs. It allows you to select and codegen only those parts of a program which most effect the speed of execution. The remaining parts of the program can be left in p-code form. This mixed mode operation allows you to increase execution speed without dramatically increasing program size.

A. When to Use Mixed Mode

The use of mixed mode is usually not important until you start developing large programs. Small programs can be totally codegened without the size increase becoming a significant factor. However, completely codegening large programs (>1000 lines) may cause a size increase which will prevent the program from being executed. The code size of the program can become so large that there is no longer enough room for data storage. This of course depends on the data storage requirements of the program.

When developing large programs, you should not consider code generation until after executing the program in p-code form. Observe the execution speed to determine whether or not it is adequate for your application. If not, the next step is to decide what areas of the program are most effecting the speed. Long loops are typical areas of a program where most of the execution time is spent. Another area might be a low level procedure or several procedures which are called frequently throughout a program. After deciding which areas of the program are effecting execution speed the most, separate them from the rest of the program and codegen them. The selection and separation process is easiest if the program is well modularized. That is, the program is already segmented into modules, each performing a distinct and well defined function.

B. How to Use Mixed Mode

Once the particular areas of a program have been selected for codegening, they must be separated from the rest of the program. Any selected modules (procedures and/or functions) should be declared as externals. (See the Pascal Reference Manual) The separated areas should then be compiled separately from the remainder of the program. The compiler nullbody option is required to compile procedures and/or functions which are separated from the main program body. Once compiled, the selected areas may be codegened and then linked to the remainder of the program using the linking loader. Once linked, a command file can be built using the BUILD command of the linking loader.

NOTE: There is an alternative way of separating modules of a program without separating them in the Pascal source. The p-code object (/OBJ) file can be split. (See chapter 5)

C. Example of Mixed Mode Operation

The process for mixed mode operation is summarized in the following list of steps.

1) Select the areas which most effect program speed.
2) Separate the selected parts of the program from the remainder of the program. Any selected procedures or functions should be declared as EXTERNAL in the main program. Place the separated modules in a separate file or files. Use the nullbody compiler option to put the separated modules in a form suitable for the compiler.
3) Compile all parts of the program.
4) CODEGEN the parts of the program which were selected to increase execution speed.
5) LINKLOAD all compiled parts of the program together and build an executable command file.

The following example demonstrates this process. The program used for this demonstration does not perform any useful function, but merely demonstrates mixed mode operation. In reality, a program of the size demonstrated should be totally translated to machine instructions rather than using mixed mode. The size increase due to code generation is insignificant for such small programs.

Program listing

```
(*$NO INOUT*)
PROGRAM MIXED_MODE;
TYPE    ALPHA = ARRAY(.1..8.) OF CHAR;
FILENM = ARRAY(.1..72.) OF CHAR;
VAR     FN  : FILENM;
ID,T  : ALPHA;
OUTPUT :TEXT;

PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
PROCEDURE SET$ACNM(VAR F : TEXT;VAR FN : FILENM; LEN : INTEGER;
                   ID : ALPHA); EXTERNAL;

PROCEDURE LOOP;
(* MODULE TO BE CODEGENED *)
VAR CALCULATION,I : INTEGER;
BEGIN
  FOR I:=1 TO 10000 DO
  BEGIN
  CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
  END
END;  (* END LOOP *)
BEGIN            (* MAIN PROGRAM *)
  (* DIRECT OUTPUT TO THE SCREEN *)
  FN(.1.):=':'; FN(.2.):='C';
  ID:='OUTPUT  ';
  SET$ACNM(OUTPUT,FN,2,ID);
  REWRITE(OUTPUT);
  TIME(T);
  WRITELN(OUTPUT,'STARTING TIME : ', T);
  LOOP;
  TIME(T);
  WRITELN(OUTPUT,'FINISHING TIME : ', T);
END.      (* END PROGRAM *)
```

Step 1) Select the areas effecting execution speed the most.

Examining the above program, you can see that the procedure named LOOP contains a very long FOR loop (1 to 10000). Inside this loop is a long calculation. Any long loop containing a significant number of statements or calculations will benefit substantially from code generation. The procedure LOOP is where the majority of program execution time is spent. Therefore, it is a good choice for code generation.

Step 2) Separate the selected modules from the rest of the program, declaring them as externals in the main program and putting them into a form suitable for compiling.

The following listing shows the procedure LOOP separated from the main program. It is declared as an external procedure within the main program and the compiler nullbody option is used to turn the procedure into a valid Pascal program. The main program and the procedure LOOP must be placed in separate files. For example, the main program could be placed in a file named MAIN/PCL and the procedure placed in a file named LOOP/PCL.

```
(*$NO INOUT*)
 PROGRAM MIXED_MODE;
 TYPE    ALPHA = ARRAY(.1..8.) OF CHAR;
 FILENM = ARRAY(.1..72.) OF CHAR;
 VAR     FN  : FILENM;
 ID,T  : ALPHA;
 OUTPUT :TEXT;

 PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
 PROCEDURE SET$ACNM(VAR F : TEXT;VAR FN : FILENM; LEN : INTEGER;
                    ID : ALPHA); EXTERNAL;

 PROCEDURE LOOP; EXTERNAL;

 BEGIN             (* MAIN PROGRAM *)
   (* DIRECT OUTPUT TO THE SCREEN *)
   FN(.1.):=':'; FN(.2.):='C';
   ID:='OUTPUT  ';
   SET$ACNM(OUTPUT,FN,2,ID);
   REWRITE(OUTPUT);
   TIME(T);
   WRITELN(OUTPUT,'STARTING TIME : ', T);
   LOOP;
   TIME(T);
   WRITELN(OUTPUT,'FINISHING TIME : ', T);
 END.

 ------------------------------------------------

 PROGRAM SEPARATE_COMPILATION;
 PROCEDURE LOOP;
 (* MODULE TO BE CODEGENED *)
 VAR CALCULATION,I : INTEGER;
 BEGIN
   FOR I:=1 TO 10000 DO
     BEGIN
     CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
     END
   END;  (* END LOOP *)

 BEGIN                        (* MAIN PROGRAM *)
   (*$NULLBODY*)
 END.
```

Step 3) Compile all parts of the program.

```
TRSDOS Ready
PASCAL MAIN

TRS80 PASCAL  VER: 02.00.00 14000000   00:01:28 05/03/83    PAGE   1
-----------------------------------------------------------------------
   1|(*$NO INOUT*)
   2|PROGRAM MIXED_MODE;
   3|TYPE    ALPHA = ARRAY(.1..8.) OF CHAR;
   4|        FILENM = ARRAY(.1..72.) OF CHAR;
   5|VAR     FN  : FILENM;
   6|        ID,T  : ALPHA;
   7|        OUTPUT :TEXT;
   8|
   9|PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
  10|PROCEDURE SET$ACNM(VAR F : TEXT;VAR FN : FILENM; LEN : INTEGER;
  11|                   ID : ALPHA); EXTERNAL;
  12|
  13|PROCEDURE LOOP; EXTERNAL;
  14|
  15|BEGIN                          (* MAIN PROGRAM *)
  16|  (* DIRECT OUTPUT TO THE SCREEN *)
  17|  FN(.1.):=':'; FN(.2.):='C';
  18|  ID:='OUTPUT  ';
  19|  SET$ACNM(OUTPUT,FN,2,ID);
  20|  REWRITE(OUTPUT);
  21|  TIME(T);
  22|  WRITELN(OUTPUT,'STARTING TIME : ', T);
  23|  LOOP;
  24|  TIME(T);
  25|  WRITELN(OUTPUT,'FINISHING TIME : ', T);
  26|END.
NO ERRORS DETECTED

TRSDOS Ready
PASCAL LOOP

TRS80 PASCAL  VER: 02.00.00 14000000   00:01:28 05/03/83    PAGE   1
-----------------------------------------------------------------------
   1|PROGRAM SEPARATE_COMPILATION;
   2|PROCEDURE LOOP;
   3|(* MODULE TO BE CODEGENED *)
   4|VAR CALCULATION,I : INTEGER;
   5|BEGIN
   6|  FOR I:=1 TO 10000 DO
   7|    BEGIN
   8|    CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
   9|    END
  10|END;  (* END LOOP *)
  11|BEGIN                          (* MAIN PROGRAM *)
  12|  (*$NULLBODY*)
  13|END.
NO ERRORS DETECTED
```

Step 4) CODEGEN the parts selected to increase speed.

```
TRSDOS Ready
CODEGEN LOOP
LOOP

STACK USED = 15915 OF xxxxx  HEAP USED = 882 OF xxxx
```

Step 5) LINKLOAD all compiled parts of the program and build an executable command file.

The executable program will be placed in file MIXED/CMD.

```
TRSDOS Ready
LINKLOAD
L=LOAD, R=RUN, T=TRSDOS, I=INIT, S=SYMBOLS, B=BUILD CMD
>> L
FILE      = MAIN/OBJ
MIXED_MO
xxxxx BYTES LEFT
>> L
FILE      = LOOP/COD
LOOP
xxxxx BYTES LEFT
>> L
FILE      = TRSLIB/OBJ
SETCSR
GOTOXY
GETKEY
INKEY
CLEARSCR
CLEARGRA
WRITECH
WRITESTR
INP
GET$PROC
IO$ERROR
HP$ERROR
TIME
DATE
ITIME
SETPOINT
RSETPOIN
TESTPOIN
USER
CALL$
$MEMORY
NOBLANK
READCURS
PEEK
POKE
INIT$FIL
```

```
FILE$STA
SET$ACNM
xxxxx BYTES LEFT
>> B
STACK SIZE :
FILE     = MIXED/CMD
```

The program may now be executed by typing MIXED.

```
TRSDOS Ready
MIXED
STARTING TIME : 00:02:48
FINISHING TIME : 00:02:51
```

The execution time spent inside the LOOP procedure may be calculated by subtracting the starting time from the finishing time. With the LOOP procedure codegened, the execution time is 3 seconds. The same program with the procedure LOOP not codegened executes in 14 seconds, a factor of 4.7 difference in execution speed. This may be tested by linking the file LOOP/OBJ instead of the file LOOP/COD and running the program over.

SYSTEM OVERVIEW

The TRS-80 Pascal Compiler is an 8500 line Pascal program which has itself been compiled into a very compact p-code form. The p-code form of the compiler has further been reduced in size by the optimizer supplied with the ADP. The optimization was necessary in order to make the compiler run in a 48k system. The p-code form of the compiler was reduced in size by approximately 28%, from 39k down to 28k.

For larger systems, where more memory is available, selected parts of the compiler have been translated to machine instructions by the code generator which is also supplied with the ADP. The code generator translates p-code instructions to native machine instructions for the purpose of increasing execution speed. Since code generation also increases size, only those sections which effected execution speed the most were translated to machine instructions.

The ADP provides the tools that were essential in the development of the compiler. These tools provide the same capability in the development of application programs.

A. The Pcode

The p-code generated by the compiler was specifically designed for the Pascal language. The p-code resembles an assembly language for a stack machine. The p-code was designed to efficiently implement Pascal functions. Therefore, each p-code instruction performs a much more complex function than a machine instruction. In fact, a p-code instruction is equivalent to an assembly language subroutine. This is the reason that p-code is so much more compact than native machine code.

B. The Interpreter

The interpreter is a highly optimized assembly language program whose purpose is to interpret p-codes. Since the computer hardware cannot understand p-code instructions, the interpreter is necessary to execute programs which have been compiled into p-code instructions. The interpreter can be thought of as a processor whose instruction set is the set of p-codes. The interpreter has the ability to switch between p-code and machine code. A particular p-code instruction tells the interpreter that native machine instructions follow. The interpreter then points the program counter (PC) register to the first of the native machine instructions and the hardware

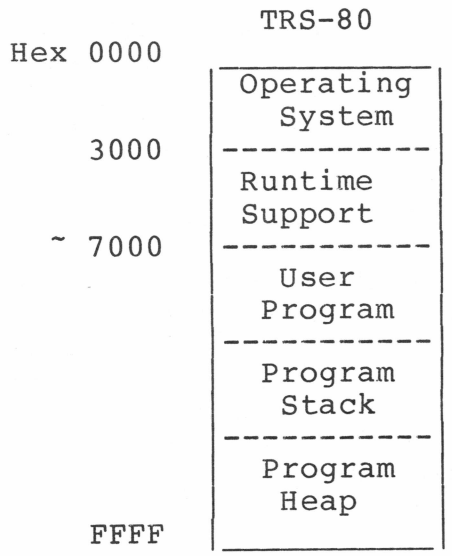
begins executing instructions. The ability of the interpreter to switch between p-code and machine code allows programs to contain mixed instructions. This means that parts of a program may be codegened for speed while the remainder of the program is left in p-code form for compactness.

C. The Runtime Support

The runtime support consists of the interpreter, a loader, a routine to set up the Pascal stack and heap, and all the input/output (I/O) routines. When building command files with the linking loader, all the runtime support is included with the program being built. Therefore, the total size of an executable program is determined by adding the size of the runtime to the size of the object program. The object program may be p-code, native code, or a mixture of both. The size of the object also includes any libraries which are linked, such as the string library.

D. The Memory Map

The following diagram shows the layout of memory useage by the Pascal system. The runtime area is approximately 16K bytes long. The memory remaining after subtracting off the operating system, the runtime, and the program area is allocated to stack and heap. This is the data area for the program. The stack is used for storing the programs static variables. The size of the stack is specified at the time the program is run (using the RUNP command) or built (using the LINKLOAD command). The remainder of memory is allocated to the heap which is used for storing dynamic variables.



E. How the Optimizer Works

The optimizer is a program which contains a loader for loading p-code object files. The loader loads and operates on one module (procedure and/or function) at a time, maintaining context as it operates on each individual module. The p-code instructions are analyzed to determine whether or not they may be compressed into shorter instructions.

Since the compiler is one pass, it must generate some branch and addressing instructions without knowing the actual displacements. This makes it necessary to allocate two byte operands for unknown displacements in order to handle all cases. However, in many cases the displacements can be specified using only one byte. The optimizer looks for such cases and compresses the p-code instructions in order to take advantage of the need for only a single byte operand.

The optimizer also looks for other types of situations where compression of instructions is possible. For example, all multiply by two instructions are converted to add instructions. In certain cases, consecutive instructions can cancel one another out (eg. an increment followed by a decrement). The optimizer eliminates such cases. The optimizer also performs constant folding (ie. it replaces arithmetic operations involving only constant operands with a single constant value). For example, 2+2 would be replaced by the single constant 4.

G. How the Code Generator Works

The code generator is a program which contains a loader for loading p-code object files. The code generator loads one module (procedure or function) at a time and translates the p-code instructions to machine instructions. As noted earlier, a p-code instruction is equivalent to several machine instructions, so the translation process will increase the total number of instructions in the object (/COD) file.

There are a few p-code instructions which perform very complex functions. To perform equivalent functions in machine code would require a very large number of instructions. Therefore, a few selected p-code instructions are not translated to machine instructions. They are left in p-code form and executed as subroutine calls to assembly language routines within the interpreter. Handling complex functions in this manner prevents the /COD file from becoming as large as it would with complete translation.

SYSTEM OUTPUT

A. Assembly Language

The native code generator has the capability of producing assembly language source in addition to object code. It is not necessary in normal circumstances to assemble the source, since the object code emitted by CODEGEN is exactly equivalent to the result of assembling the source. The assembly language is provided as a means for the programmer to examine the code produced by the native code generator. In some cases, the programmer may wish to optimize this code by hand and assemble it. It is expected that the need to do this will be rare, since the effort is substantial and the improvements that can be made are minor. If you wish to assemble the source output of CODEGEN, then the Alcor Systems multiprocessor assembler is required.

The source output of codegen is useful to the assembly language programmer who wishes to link assembly language modules to Pascal and to call them as Pascal procedures or functions. A possible technique to accomplish this is to write a Pascal procedure or function with the same name and calling sequence as the assembly language routine. The actual code can be left out and perhaps replaced by a template that merely accesses the parameters that will be used in assembly language.

The dummy procedure produced above can be compiled by Pascal and run through the code generator with the source option enabled. Pascal and codegen will generate the proper Pascal procedure or function linkage and will calculate the addresses of variables and parameters referenced in the body of the procedure. The generated code can then be used as a skeleton for the assembly language that actually implements the functions required.

B. Assembly Language Structure

The assembly language code emitted by CODEGEN is designed for assembly by the Alcor Systems multiprocessor assembler. This assembler provides the features required to support Pascal and the ability to mix Z80 code (or 6502 code, or 1802 code, or 8080 code) with P-code. Essential assembler features include the ability to switch among target processors (Z80 to P-code), the ability to define and reference external symbols (externals are resolved at link edit or load time), and the ability to generate p-code addressing modes (program counter relative, stack displacement, access to common blocks).

Each Pascal procedure or function forms a separate module. All symbols, labels, and instructions are local to the module and reference other modules only via explicit external references. Modules begin with a module identification. For Pascal generated code, the module name is the name of the procedure or function truncated to 8 characters. Each procedure or function also contains an external definition of the procedure name. This is signaled with the "DEF" assembler directive. The DEF statement causes the name and its value to be defined externally, so that other modules can call it.

Switching between modes (native vs. p-code) takes place within the procedure. Some operations performed by Pascal are sufficiently complex that they are implemented with subroutines. Inclusion of the actual code in-line would make the generated code unreasonably large. When these operations (such as input, output, or set operations) are performed, the code generator produces a call to a runtime procedure. These runtime procedures are already part of the Pcode interpreter. Rather than reference them again (and require another copy), the processor is switched back to p-code mode and the interpreter is allowed to perform the operation.

When in mixed mode, all procedure calling is performed using the p-code interpreter. Since code for each module is separate, and since modules may be split before being loaded, it is unknown whether the procedure being called is p-code or native code. Therefore, every module is entered in p-code mode. If the module is native code, the processor is switched to native mode immediately after entry to the procedure.

C. Assembly Language Format

The native code emitted by CODEGEN uses extended 8080 mnemonics. This is done primarily for historical reasons and since the 8080 instruction set more clearly distinguishes instructions by format. Use of 8080 extended mnemonics affects only the source output of codegen, as the Z80 instruction set is used and converted directly to object code by codegen. Each instruction occupies one line. Labels are left justified and begin with a letter. Each instruction has an op-code which is either an 8080 instruction or a Z80 instruction. There are also pseudo-operators (pseudo-ops) that provide instructions to the assembler rather than generating code.

Operands use standard register names. In many cases, the names of the Z80 index registers are merged with the op-code (e.g. PUSHIX pushes the IX index register). This simplifies interpretation by the assembler. Operands may also use symbolic labels and constants. Constants are normally expressed in hexadecimal (base 16) with a leading greater than sign (">") to specify hexadecimal to the assembler.

Pseudo-operators

IDT	identifies the module and gives it a name
EQU	defines the value of the label to the result of evaluating the operand
DEF	defines the operand as an external symbol
REF	specifies that the operand is an external symbol that is defined in another module
CSEG	Specifies the name and size of a common block
QLIST	Selects the compact format for the assembler listing
END	Signals the end of the module
ENTRY	Defines an entry point into the module
SETCPU	Selects the processor whose assembly language is being assembled

D. Object Format

TRS-80 Pascal uses its own format for object code. The main reason for this is that support for many of the features of TRS-80 Pascal are not present in existing object formats. For example, TRS-80 Pascal supports common blocks for statically allocated variable storage and the object format must in turn allow for this.

The p-code generated by the compiler is address independent. That is, it contains no absolute memory addresses and can execute without change when loaded anywhere in memory. All branching and calling of procedures within the pcode is done relative to the current program counter. Since procedures are compiled into separate modules, calculation of these relative addresses must be done when the code is loaded. The object format supports external references that are program counter relative.

The object code is tagged hexadecimal and is emitted in a line oriented stream that is compatible with a Pascal text file. In particular, the object code is character oriented and contains only printable ASCII characters. This allows the object to be manipulated by text editors or transmitted over modems. This is not possible with bit oriented formats.

Each item in the object file begins with a tag which is usually an upper case letter. The tag defines the type of item and the number and size of the fields to follow. Tags are followed by one or more fields that specify the information to be loaded. Three types of fields exist. Bytes are specified with a two character hexadecimal number. Words consist of a four character hexadecimal number with the most significant byte first. Labels consist of eight character names that are the names of external symbols, common blocks, etc...

Following is a table which lists all the tags used in an object file. All tags are followed by one to three fields of information, each field being either a byte, word, or label. The meaning of each tag is also shown.

Tag	Field1	Field2	Field3	Meaning
A	byte			Absolute(non-relocatable byte)
E				End of module
F				End of line
G	word	label		Definition of external symbol
I	word	label		External reference declaration
J	label			Module name
Q	word			Reference to external symbol
M	word	word	label	Definition of common block
N	word			Reference to common
O	word	word		Overlay definition
P	word			Code (PC) origin
K	word			Relative reference to external
W	word			Relocatable word
X	word			Absolute word
Y	word			Entry point definition
:				End of file

E. Splitting Object Modules

Since object files are in ASCII format, they may be edited with a text editor or used as input to a Pascal program. The following is a list of the pure pcode output (/OBJ) file for the LOOP procedure in the mixed mode operation example. Following it is a listing of the object (/COD) file which results from running the pcode object through the code generator. As you can see, the code generation has caused approximately a factor of 2 increase in size.

Pure Pcode Listing (/OBJ)

```
JLOOP    P0000G0000LOOP    A01X0000A38A02A03X0001A15A04A10A04A03X2710A07F
A15A06A2BA4EX0000A03X0001A03X0002A22A03X0003A22A03X0004A22A03X0005A22A03F
X0006A22A03X0007A22A03X0008A22A03X0009A22A03X000AA22A03X000BA22A03X000CF
A22A03X000DA22A03X000EA22A03X000FA22A15A02A10A04A30X0004A10A06A27A21AB9F
P0014X0047P005DA3AP0001X0006E
:
```

Native Code Listing (/COD)

```
JLOOP    G0003LOOP    AC1AEBAE9A01X0006A38A02A55A21X0001ADDA75A04ADDA74F
A05ADDA6EA04ADDA66A05AE5A21X2710AE5ADDA75A06ADDA74A07AC1AE1A78AACAEDA42F
A28A09A47A3FA1FAA8A07AE6A01A18A02A3EA00AA7AC2X0000A21X0001AE5A21X0002AC1F
A09AE5A21X0003AC1A09AE5A21X0004AC1A09AE5A21X0005AC1A09AE5A21X0006AC1A09F
AE5A21X0007AC1A09AE5A21X0008AC1A09AE5A21X0009AC1A09AE5A21X000AAC1A09AE5F
A21X000BAC1A09AE5A21X000CAC1A09AE5A21X000DAC1A09AE5A21X000EAC1A09AE5A21F
X000FAC1A09ADDA75A02ADDA74A03ADDA6EA04ADDA66A05AE5ADDAE5AE1A01X0004A09F
A4EA23A46A03A70A2BA71ADDA6EA06ADDA66A07AC1AAFAEDA42A20A01A3CAA7ACAW003AF
P0038W00BDP00BDACDW0000A3AE
:
```

Each module in an object file begins with the module name. Therefore, it is possible to split a file containing several modules into several files, each containing one module. This is an alternate method of segmenting large programs where it is desired to perform code generation on only selected parts.

There are two ways to split the object modules. One is to text edit them. The other more desireable method is to write a Pascal program to split them. A simple program may be written to read the pcode (/OBJ) file. Each time a module is encountered, open a file of the same name as the module and write the module to that file. Once all the modules are separated into different files, selected modules may be input to the code generator and translated to native machine instructions. The linking loader may then be used to link the individual modules and build an executable command (/CMD) file.

MASTER CROSS REFERENCE INDEX

B = Beginners Guide E = Editor Manual R = Reference Manual
S = System Guide T = Tutorial A = Advanced Development Package

ABS	R73, T36, T39
ACCESS	R21
ADD	T21
ADDRESS	A22
AND	R46
APPEND	E 6, E36
ARCTAN	R73
ARITHMETIC	A18, R44, T 4, T17, T19
ARRAY	R28, T41
ASCII	B 7, A22, E13,R102
ASSEMBLY	S19, S20, S37, A 7, A16, A19
ASSIGNMENT	R54, T17
AUTO-INDENT	E24
BACKUP	B11, B12, E37, E40
BEGIN	R23, R55, T 6, T21
BLDSTR	R108
BLOCK	R12, R13, R16, T37
BLOCK CMDS	E17, E34
BOOLEAN	R25, R52, T 9
BRANCHING	R63, T28
BUILD	S11, A10
BYTE	A18, A22
CALL$	S20
CASE	R60, T24, T26
CHAR	R25, T 9, T11
CHARACTER	S36
CHR	R25, R74
CLEARGRAPHIC	S27
CLEARSCREEN	S27
CLOSE	R86
CLOSERAND	S32
CMDLINE	S19
CMPSTR	S36
CODEGEN	S 3, A 1, A 5, A18
CODEINIT	A 7
COMMAND MODE	B10, B11, E 8, E13, E24
COMMENT	R10, T40
COMMON	S10, A19, A22, A23, R20,R119
COMPARE	R45, T27

COMPILE	B13, S 5, S 6, S 7
COMPILER	A16
COMPONENT	R35, T41, T46, T48
COMPOSE MODE	E 7
COMPOUND	R55
CONC	S36
CONDITIONAL	R58, T30
CONST	R18, T11, T13
CONSTANTS	A20, R 8, R 9, R18
COS	R73
COUNTER	R56, R89
CPYSTR	S36
CURSOR	B 8, B 9, E14, E15, E27, T 7
DATABASE	S 8, T63
DATE	S18
DECLARATION	R16, T 9, T11
DECODED	S35
DECODEI	S35
DECODER	S35
DEFINE MACRO	E45, E47
DEFINED	S10
DEFINITION	A20, A23, R16
DELETE	S36
DELETE TABS	E24, E41
DELFILE	S24
DELIMITERS	R10
DEVICE NAMES	B16, S13
DIMENSION	R28, T42, T51
DISPOSE	R42
DIV	R44, T18, T21
DO	R57
DOUBLE	R88
DOWNTO	R56, T24, T26
DYNAMIC	S14, A17, R40, R41, R42, T52
EDIT	B 7, E12, E42
ELSE	R58, T28, T31
ENCODED	S35
ENCODEI	S35

ENCODER	S35
END	R23, R55, T 6
ENUMERATED	R26, T45, T48
ENUMERATION	R25
EOB	B 7, E 6, E13
EOF	E 6, R73, R76, R77, T14
EOLN	R34, R73, R76, R77, T14, T34
ERROR	B14, B17, B18, B19, S12, S33, R98
ESCAPE	R65, R75
EXIT	B11, B12, E37
EXP	R73
EXPRESSION	R49, T 4, T17, T22, T24
EXTENSIONS	R105
EXTERNAL	S17, A10, A20, A23, R69, R70
EXTMEM	S29
FIELDS	A22, R34
FILE	R32, R33, R34, R76, T 7, T11, T14
FILE$STATUS	S23
FILE CMDS	E36
FILE NAMES	B16, S13
FIND	S36
FOR	R56, T23
FORDECL	R89
FORWARD	R68, T39
FUNCTION	S17, S35, R22, R65, T32, T36
GET	R112
GETKEY	S23
GETSTR	R110
GLOBAL	R66, R69, R70, T34, T38
GOTO	R17, R63, T28
GOTOXY	S27
HB	R74
HEADING	R13, R15
HEAP	B13, S 8, S11, S15, S20, A17, R41
HELP	E18, E41
HEXADECIMAL	S 8, S11, A20, A22, R 9, R83
HP$ERROR	S20
HSCROLL	E28
IDENTIFIER	R 7, T 7, T36, T38
IF	R58, R90

IN	R30, R46
INCLUDE	R92
INIT	S12
INKEY	S23
INOUT	A12, R89
INP	S22
INPUT	A 3, A 5, R33, R76, T14, T16
INSERT	S36
INSERT MODE	B10, E13, E24
INTEGER	R 8, R24, T10
INTERPRETER	A 1, A16, A17
INTERSECTION	R32, R44
IO$ERROR	S24
KEYS	B 8, E13
KEYWORD	R 9
LABEL	A21, A23, R17, R53, R63
LB	R74
LEFT$	S35
LEN	S35
LIBRARY	S17, S35, A17, R65, R69, R91
LINE CMDS	E31
LINE NUMBERS	E25, S 8
LINKED	R21, R65, R91, T54, T62
LINKLOAD	S 4, S 9, A10, A17
LIST	A 3, R93
LISTING	B13, S 8
LITERAL	R83
LN	R61, R73
LOAD	S 9, S10
LOADER	A10, A18
LOCAL	R66, R67, R71, T34, T38
LOCATION	R74
LOGICAL	R25, R32, R76
LOOP	R55, R56, R57
MAXINT	R18
MEMBERSHIP	R30, R46, T59
MEMORY	S15, S16, A 2, A 6, A16, A17, A22, E 6
MESSAGE	R72, R76, R87
MID$	S35
MIXED	A 9, A10, A11, A20, A23, R44
MOD	R44, R47, R51, R58, T18, T21

NATIVE	A 5, A16
NESTED	R12, R66
NEW	R41, R42, T52, T54, T56
NIL	R42, R59, T55
NOBLANK	S27
NOT	R46, R52, T27
NULLBODY	A12, R70, R91
NUMBERS	R 8
OBJECT	S 5, A 6, A22, A23, A24
ODD	R73
OF	R28, R37, R60
OPEN	R77, R78
OPENRAND	S31
OPERATOR	A20, R44, T18
OPTIMIZER	S 2, S 3, A 1, A 2, A 6, A16, A18
OPTIONS	R88, T 1
OR	R46, R52
ORD	R22, R25, R74, T33
ORDINAL	R24
OTHERWISE	R60, R61
OUT	S22
OUTPUT	R33, R76, T 6, T 8, T14
OVERLAY	A23
OVERVIEW	S 2, A16
OVERWRITE	E13
PACK	R28, R74
PACKED	R28, R38
PAGE	R87
PAGESIZE	R94
PARAMETER	T 4, T32, T34, T40
PARAMETERS	E22, E27, R13, R14, R15, R50, R63
PARENTHESES	R47
PASCAL	S 3, S 5, S 6, A16, T 2
PASCALB	S 4
PCODE	A 1, A 2, A 5, A 6, A 9, A16, A18, A19
PEEK	S21
POINTER	R40, R41, R42, R43, T52, T54, T56
POKE	S21
PRECEDENCE	R47, T17, T19, T22, T27

PRECISION S16
PRED R75
PREDECLARED R72
PREDEFINED R18, R24, R25, R26, R33, R42, T 3, T 9, T11, T16, T26, T36, T44, T46, T59
PRINTER S 6
PROCEDURE R13, R22, R63, R65, T 3, T32, T34, T36
PROGRAM R12, R13, T 6
PTRCHECK R97
PUT R112
QUIT B11, E38
QUOTE E22, R 9
RANDOM S31, S32, S33
RANGECHK R96
READ R79, T14, T40, T42, T48, T50
READCURSOR S28
READLN R34, R76, R84, T14, T42
READRAND S31
REAL S16, R 8, R27, T 3, T 9, T11, T20, T26
RECORD R34, T 3, T 5, T46, T48
RECURSION R71
RECURSIVE R71, R90, T 5
REFERENCE A20, A23
REGISTER A20, S19, S20
RELATIONAL R45, R46, T 4, T27, T29, T59
RENAME S24
REPEAT R57, R58, T 4, T30
REPETITIVE R55
REPLACE S36
RESERVED R 9
RESET R72, R76, R77, R78, T14
REWRITE R72, R76, R78, R79
RIGHT$ S35
ROLL E26
ROUND R74
RSETPOINT S28
RUN B15, S 5, S 7, S 8, S11
RUNTIME A17
SCOPE R66, R68, R69, T37, T39, T49
SELECTOR R38, T24, T26
SEMICOLON R11

SAVE E40
SCROLL E14, E27, E28
SEPARATE A 9, A11, A22
SET R28, R29, R30, R31, R32, T 3, T 5, T59, T61
SETACNM S26
SETEDIT E 5, E10
SETPOINT S28
SETUP/EDT B 6, E 5, E10, E11
SETUP FILE E 4, E48, E54
SET$ACNM S25
SHOWFILE E39
SHOWLINE E29
SIN R61, R73
SOUND S18
SOURCE A 8, A10, A20
SPEED A 1, A 5, A10, A15, A16
SPLIT OBJECT A10, A24
SQR R23, R73
SQRT R73
STACK B13, S 6, S 7, S 8, S14, A16, A17, R65, R71
STANDARD R105
STATEMENT R23, R53
STATE CMDS E24
STRING R 9, R108, T 7, T11, T14, T41, T43
STRING CMDS E32
STRING LIB S35
STRUCTURE T 6, T32, T36
STRUCTURED R12, R28
STR$ S35
SUBRANGE R27
SUBROUTINES T 3, T32
SUBSET R27, R31, T59
SUBTRACT T21
SUCC R75, T44
SUPERSET R31, T59
SVC S18
SYMBOLS S10, A21, A23, R10
SYNTAX R 5
TAB E16, E27, E28
TABIFY E25
TAB SET E26, E41
TABLE A22
TAG A22, A23
TERMINAL B 6, E 5, E10
TESTPOINT S28
TEXT A22, R33, R34, R76, R79, R83, R84, R85, T 8, T13, T15, T48, T50
TEXT BUFFER B 7, E 6
THEN R58
TIME S18
TO R56

TRANSLATE	E43, E44
TRUNC	R74, T26
TYPE	R19, R24, R27, R28, R40, R48, T 3, T 9, T11
UNARY	T18
UNION	R31, R44, T59
UNPACK	R74
UNTIL	R57, T30
USER	S19
VAR	R20, T 9
VARIANT	R35, R37, R38, R39
WHILE	R57, T21, T29, T31
WIDELIST	R95
WITH	R61, R62, T49
WORK FILE	E 6, E 7
WRITE	E 6, E40, R76, R81, T 6, T14
WRITECH	S22
WRITELN	R34, R76, R85, T 6, T13, T15
WRITERAND	S32
WRITESTRING	S22
$MEMORY	S20

RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAD MOUNT DRUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

10/83-TM

Printed in U.S.A.